Universität Ulm
Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Institut für Organisation und Management von Informationssystemen

Ein hierarchisches Scheduling-Modell für unbekannte Anwendungen mit schwankenden Ressourcenanforderungen

Dissertation

zur Erlangung des Doktorgrades

Dr. rer. nat.

der Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
der Universität Ulm

vorgelegt von

Vladimir Volodiev Nikolov

aus Sofia, Bulgarien

April 2016

Amtierender Dekan: Prof. Dr. Tina Seufert
1. Gutachter: Prof. Dr.-Ing. Franz J. Hauck
2. Gutachter: Prof. Dr.-Ing. Stefan Wesner

Tag der Promotion: 04. August 2016

Zusammenfassung

Weiche Echtzeitsysteme, die mit unbekannten, dynamisch gestarteten und gestoppten Anwendungen umgehen müssen, sind schwer zu analysieren in Hinblick auf ihre Planbarkeit. Ein solches System muss in der Lage sein, sich dynamisch an die tatsächlichen Anforderungen der Anwendungen anzupassen. Es benötigt ebenso Mechanismen zur Kontrolle und Behebung von Überlast, welche durch die Konkurrenz der Anwendungen um gemeinsam genutzte Ressourcen entstehen kann. Da Schätzungen der Ausführungszeiten von Anwendungsaktivitäten von vornherein nicht bekannt sind, muss das System diese zur Laufzeit ermitteln und sich auf Schwankungen im Ressourcenverbrauch der Aktivitäten einstellen. Im Rahmen dieser Promotionsarbeit wurde aus diesem Kontext ein adaptives, hierarchisches Scheduling-Modell abgeleitet, das mit einer dynamischen Menge von unbekannten Applikationen mit schwankenden Ressourcenanforderungen umgehen kann. Auf der untersten Ebene stellt der Scheduler die korrekte Ausführung von Applikationsaufgaben, gemäß ihrer zur Laufzeit approximierten Bedarfe, sicher. Auf den höheren Ebenen werden Applikationen voneinander isoliert, während der Scheduler kontinuierlich die ihnen zugewiesenen Kapazitäten kalibriert und lokale Planbarkeitsanalyse betreibt. Entstehen Engpasssituationen, so wird ein Mechanismus zur gezielten Herabsetzung (Degradierung) der Ressourcenanforderungen der Applikationen angewendet. Dabei wird dynamisch jeweils eine von mehreren Qualitätsstufen (Betriebsmodi) der Anwendungen ausgewählt und umgeschaltet. Solche Rekonfigurationen sollen allerdings so selten wie möglich stattfinden, da sie mit zusätzlichen Kosten für die Anwendungen und das System verbunden sind. Mit dem erarbeiteten Scheduling-Modell wird das Problem des Planbarkeitsnachweises räumlich partitioniert und zeitlich verteilt. Planbarkeitstests werden nur bei Bedarf für bestimmte Bereiche im System betrieben, dann und dort, wo es notwendig ist. In der Zwischenzeit nimmt der Scheduler die Planbarkeit der bestehenden Konfiguration an, solange die Kostenschätzungen der Applikationsaktivitäten bestimmte adaptive Sicherheitspuffer nicht überschreiten. Werden die Toleranzgrenzen verletzt, so wird eine neue optimale Ressourcenkonfiguration berechnet und dabei unter Umständen die Qualität einer oder mehrerer Anwendungen verändert. Treten Frist- oder Toleranz-Verletzungen periodisch auf, so werden diese vom System detektiert und damit ver-

bundene periodische Rekonfigurationen automatisch unterdrückt. Das Scheduling-Modell wurde mit RTSJ implementiert und innerhalb eines Soft-Echtzeit-Frameworks ARTOS integriert. Für dessen Evaluation wurden unter anderem künstliche Anwendungen entwickelt, die eine reale Arbeitslast erzeugen und dabei bestimmte Lastprofile befolgen. Bei den Experimenten wurden die Applikationen zunächst mit einer niedrigen Priorität gestartet und initial ausgemessen. Dadurch entstehen erste Schätzungen für alle Anwendungsmodi, welche bei späterer Aktivierung verfeinert werden. Somit konnte bereits experimentell nachgewiesen werden, dass (a) das System schnell eine optimale Modi-Selektion für die Anwendungen etabliert, (b) es planbar bleibt, solange Tasks innerhalb der ermittelten Toleranzgrenzen verbleiben, (c) Fristverletzungen nur dann auftreten, wenn Tasks ihre Grenzen verlassen, (d) Planbarkeitstests häufiger für Apps durchgeführt werden, deren Task-Anforderungen sich stärker als die Adaptionsgeschwindigkeit ihrer Sicherheitspuffer verändern und (f) zyklische Bursts automatisch durch das System detektiert und behandelt werden. Eine automatische Anpassung der Adaptionsgeschwindigkeit der Sicherheitspuffer, anhand eines dynamisch extrahierten Verhaltensprofils für die jeweilige Anwendung, ist in Arbeit. Anhand des bereitgestellten Entwicklungsmodells für ARTOS-kompatible Applikationen, wurde darüber hinaus im Rahmen dieser Arbeit eine Video-On-Demand Anwendung implementiert, die zur weiteren Evaluation des Modells anhand eines realen Szenarios dient. Das Scheduling-Modell wurde bereits als Grundlage für die Ressourcensteuerung innerhalb einer dynamischen Cloud-Plattform verwendet [1].

Abstract

Soft real-time systems that have to deal with unknown, dynamically loaded and stopped applications are hard to analyse for feasibility. Such systems must be capable to dynamically adapt themselves to actual requirements of applications. They further require mechanisms to control and resolve overload situations, which may arise while apps compete for common shared resources. Since estimations of actual execution times of activities are not available from the start, they have to be determined by the system during runtime. The estimations must be also adopted to fluctuations within the measured resource demands of the activities. In this PhD thesis an adaptive hierarchical scheduling model for a dynamic set of applications with fluctuating resource demands has been developed. On the lowest level the scheduler ensures correct execution of application tasks according to online approximations of their demands. On the higher levels isolation between the apps is provided while the scheduler continuously re-calibrates given capacities and performs local schedulability analysis for application tasks. In bottleneck situations an optimal degradation is applied by assigning one of different quality levels to the apps based on their resource estimations and priorities. Thereby, applications are activated in different operational modes. However, such reconfigurations must happen as rarely as possible, because they introduce an additional computational overhead for the system. With the proposed scheduling architecture the problem of proving schedulability is spatially partitioned and temporally distributed. Tests are performed portion-wise only when and where necessary. The model maintains feasibility until cost estimations violate custom safety buffers and resource bounds. In such cases a new optimal resource configuration is computed changing the quality of some apps. Periodic violations are also dedected automatically suppressing recurring reconfigurations in the system. The scheduling model was implemented with RTSJ and integrated within a soft real-time framework ARTOS. For evaluation of its scheduling mechanisms artificial apps were developed, which can produce real computational overhead within a platform following a certain load-profile. In our experiments new apps were run with a low priority until their modes were initially estimated. We were able to verify that (a) the system quickly establishes an optimal modes selection respecting priorities (b) it remains feasible when tasks reside within their tolerance limits (c)

deadline misses occur only if tasks leave their safety buffers (d) feasibility analysis is performed more frequently for apps with stronger fluctuations than their buffer adaptation speed and (f) cyclic bursts can be automatically detected and treated by the system. An automatic adjustment of the buffer adaptation speed based on dynamically extracted application profiles is planned for the future. Based on the provided architectural model for multi-mode applications, an ARTOS-compatible Video-on-Demand application was implemented and used for further evaluation of the scheduler within a realistic scenario. The scheduling model has been further used as a basis for an adaptive resource-management framework within a dynamic cloud-platform [1].

Danksagungen

Zunächst möchte ich mich ganz herzlich bei Herrn Prof. Franz Hauck für die Betreuung meiner Arbeit und für die unzähligen, fördernden fachlichen Diskussionen bedanken.

Darüber hinaus gilt ein besonderer Dank Herrn Prof. Stefan Wesner und Herrn Lutz Schubert, die mir während meiner Beschäftigung am Institut für Organisation und Management von Informationssystemen die Möglichkeit gewährt haben, mich mit neuen, spannenden Themen zu beschäftigen und somit auch neue Anwendungsfelder für die in meiner Promotionsarbeit entwickelten Modelle und Mechanismen zu entdecken.

Ferner möchte ich mich bei den Entwicklern von Aicas und ganz besonders beim Herrn Marvin Blauth bedanken, die die in dieser Arbeit vorgeschlagenen Erweiterungen und Anpassungen von RTSJ in kürzester Zeit innerhalb der JamaicaVM implementiert und integriert haben.

Diese Arbeit möchte ich meiner Familie widmen, die mich stets bedingungslos unterstützt hat.

„Es ist nicht zu wenig Zeit, die wir haben, sondern es ist zu viel Zeit, die wir nicht nutzen.“

Lucius Annaeus Seneca

„Demoskopie ist die Kunst, hinterher zu begründen, warum alles ganz anders gekommen ist.“

David Frost

Inhaltsverzeichnis

Zusammenfassung	III
Abstract	V
Danksagungen	VII
Inhaltsverzeichnis	IX
1 Einführung	1
1.1 Problembeschreibung	5
1.2 Beispiel-Szenario	6
1.3 Anforderungen	7
1.4 Thematische Einordnung der Arbeit	12
1.5 Wissenschaftliche Beiträge	14
1.6 Überblick	16
2 Grundlagen und Stand der Forschung	17
2.1 Echtzeittheorie	18
2.1.1 Ablaufplanung	21
2.1.2 Planbarkeitsnachweis	23
2.2 Verwandte Arbeiten	28
2.2.1 Reservierungsbasierte Verfahren	29
2.2.2 Methoden zur Überlastbehandlung	33
2.3 Grundlagen der Video-Codierung und -Übertragung	41
2.3.1 Video-Codierung	41
2.3.2 Video-Übertragung	44
2.3.3 Gründe für Qualitätsanpassungen	46
2.4 Qualitätsanpassungen	47

2.4.1	Realisierung von Betriebsmodi	49
2.4.2	Fallstudie: Qualität eines Videosignals	51
2.4.3	Fallstudie: Dynamische Auslieferung von Inhalten	57
3	Modell für Ressourcensteuerung	61
3.1	Applikationsmodell	63
3.1.1	Bestimmung der mittleren Task-Kosten	67
3.1.2	Schätzungen der Task-Kosten	69
3.1.3	Approximation des Verhaltens von Tasks	73
3.1.4	Problem der Planbarkeitsanalyse	77
3.2	Hierarchisches Scheduling-Modell	79
3.2.1	Scheduling auf der Task-Ebene	81
3.2.2	Scheduling auf der Applikations-Ebene	87
3.2.3	Scheduling auf der System-Ebene	105
3.2.4	Globale Betrachtung des Scheduling-Modells	114
3.3	Zusammenfassung der Beiträge dieses Kapitels	115
4	Implementierung des Modells	117
4.1	Komponentenmodell und OSGi	118
4.2	Real-Time Specification for Java (RTSJ)	120
4.2.1	Echtzeit-Aktivitäten und Scheduling	121
4.2.2	Asynchrone Ereignisverarbeitung	122
4.2.3	Asynchronous Transfer of Control	123
4.2.4	Processing Groups	124
4.2.5	RTSJ als Basis für ein Echtzeit-Komponentenframework	125
4.3	Ausführungs-Plattform	126
4.3.1	JamaicaVM	126
4.3.2	Abbildung von Echtzeit-Threads innerhalb des Betriebssystems	127
4.3.3	Abbildung von Prioritäten und CPU-Affinitäten	128
4.3.4	Voll-präemptiver Linux-Kernel: CONFIG_PREEMPT_RT	130
4.4	Implementierung mit RTSJ	131
4.4.1	Realisierung von Applikationen und Applikationsmodi	132
4.4.2	Realisierung von Tasks und Task-Modi	134
4.4.3	Applikations-lokales Task-Scheduling	139
4.4.4	Implementierung und Scheduling von CBS	144
4.4.5	Optimierung und Rekonfiguration von Applikationsmodi	158
4.4.6	Implikationen der Umschaltungen von Anwendungsmodi	160

4.4.7	Überprüfung und Aktualisierung der Task-Approximationen	163
4.5	Bewertung und Erweiterungen von RTSJ	168
4.5.1	Änderungen von Task-Perioden und -Fristen	168
4.5.2	Prozess-Gruppen	172
4.5.3	Benutzerdefiniertes Scheduling von Echtzeit-Threads	174
5	Evaluation	177
5.1	Evaluations-Umgebung	178
5.1.1	Plattform-Konfiguration	178
5.1.2	Evaluations-Werkzeuge	179
5.1.3	Künstliche Anwendungen	185
5.2	Evaluation des Scheduling auf der Task-Ebene	189
5.3	Evaluation des Scheduling auf der Applikations-Ebene	192
5.3.1	Eine Task pro Anwendung	192
5.3.2	Mehrere Tasks pro Anwendung	195
5.4	Evaluation der Adaptions-Mechanismen	201
5.5	Erfassung periodischer Bursts	206
5.6	Evaluation der Optimierung	208
5.6.1	Variation der Ressourcengrenze R des Systems	210
5.6.2	Variation der Anzahl an Applikationen $ A_i $	212
5.6.3	Variation der Anzahl an Modi $ \mathcal{M}_i $ pro Applikation	213
5.6.4	Aggregation der Faktoren	215
5.7	Evaluation des Qualitätsgewinns anhand einer VoD-Anwendung	216
5.7.1	VoD-Anwendung	219
5.7.2	VoD-Szenario ohne Degradierungs-Mechanismen	223
5.7.3	VoD-Szenario mit Degradierungs-Mechanismen	226
5.7.4	Diskussion der Ergebnisse	229
6	Abschluss	231
6.1	Zukünftige Arbeiten	234
	Literaturverzeichnis	239
	Abbildungsverzeichnis	251
	Listings	255

1

Einführung

Weiche Echtzeitsysteme finden einen immer breiteren Einsatz in industriellen Lösungen und Produkten, dort wo Informationen eher zeitnah als rechtzeitig verarbeitet und bereitgestellt werden müssen. Beispiele dafür existieren in diversen Branchen, z.B. in Fahrzeugen, die zeit- und sicherheitskritische Funktionen besitzen und zusätzlich zu autarken multimedialen Systemen geworden sind, oder bei mobilen Geräten wie Smartphones, die kontinuierlich mit ihrer Umwelt (z.B. intelligenten Häusern, Städten, usw.) interagieren und kollaborieren, um zeitnah benutzerrelevante Informationen zu extrahieren bzw. bereitzustellen. Mit der zunehmenden Rechenleistung eingebetteter Geräte, werden darauf tendenziell auch immer mehr Funktionen und Anwendungen dynamisch bereitgestellt und konkurrierend ausgeführt. Ein dynamisch erweiterbarer Funktionsumfang verlangt jedoch nach adaptiven Mechanismen für Ressourcenkontrolle und Admission in den Geräten. Hinzu kommt die Schwierigkeit der exakten Abschätzung der Anwendungsanforderungen, nicht zuletzt aufgrund der Vielfalt der möglichen Ausführungsplattformen. Unterliegen Anwendungsaktivitäten zeitlichen Fristen, so muss nun dynamisch durch eine Planbarkeitsanalyse überprüft werden, ob Ressourcen für ihre fehlerfreie Ausführung garantiert werden können.

Die heutzutage stets zunehmende Vielfalt von Hardware-Plattformen stellt neue Herausforderungen für Anwendungs- und Systementwickler. In einer Welt vernetzter Geräte wächst die

Anforderung der Interaktivität und Mobilität von Anwendungen und deren benötigten Daten. Anwendungsfunktionen werden nicht zuletzt aufgrund einer besseren Interkonnektivität und Laufzeitperformanz dynamisch in unterschiedlichen Ausführungsplattformen bereitgestellt oder migriert. Dynamische Umverteilungen von Funktionen, Komponenten oder ganzer Applikationen auf unterschiedlichen entfernten Recheneinheiten, können auch im Zuge einer optimalen Lastverteilung stattfinden. Solche Szenarien findet man zum Beispiel innerhalb Cloud-basierten Systemen, wo der Ressourcenbedarf der Anwendungen von unvorhersehbaren Schwankungen in der Häufigkeit von Nutzeranfragen abhängt [2]. Diese Dynamik wird umso mehr bestärkt durch den Kontext der Anwendungsnutzer, welche, unabhängig von den verwendeten Endgeräten, ihrem Ort und Zeit, personalisierte Inhalte abrufen bzw. emittieren und neue Funktionen ausführen möchten. Dabei können Berechnungen in der Nähe von Nutzern bereitgestellt werden, um zum Beispiel ihre Kommunikationslatenz zu verringern [3, 4, 5]. Ferner kann eine Optimierung des Datentransfers insofern vollzogen werden, dass Berechnungen in Datennähe migriert werden, um die Kosten für die Datenübertragung zu minimieren [6]. Nicht zu verkennen ist auch der Aspekt der Kostenoptimierung, wobei Anwendungen und ihre Aktivitäten innerhalb Umgebungen bereitgestellt werden, in denen die Kosten für die benötigten Rechenzyklen für bestimmte Funktionen geringer ausfallen [2, 7]. Da dies von der Beschaffenheit der jeweiligen Anwendung und ihrer Berechnungen abhängt und auch von der unterliegenden Hardware-Plattform, bezieht sich hier eine Kostenoptimierung auch auf die Auswahl der geeignetsten Plattform für die jeweilige Anwendung innerhalb einer heterogenen Umgebung [8, 9].

Die Diversität der potentiellen Ausführungsplattformen für Anwendungen erstreckt sich heute von personalisierten Geräten, wie Smartphones, Tablets und Desktop-PCs, über Rechenzentren mit leistungsstarken Server-Systemen bis hin zu eingebetteten Geräten mit speziellen Mikrokontrollern und FPGAs. Diese Hardwareplattformen unterscheiden sich nicht nur in ihrer Rechenleistung, sondern auch generell in ihrer zugrunde liegenden Beschaffenheit, Architektur, Instruktionsmenge und Einsatzfeld. Die Spezialisierung einer bereitzustellenden Funktion für eine bestimmte Zielplattform ist nicht die einzige Herausforderung für die Anwendungsentwickler und die entsprechende Laufzeitumgebung. Unterliegen Anwendungsaktivitäten zusätzlich noch zeitlichen Anforderungen und Fristen, so besteht eine weitere Herausforderung darin, die zuverlässige und reibungslose Anwendungsausführung im Voraus zu bestimmen und zur Laufzeit zu garantieren. Dafür sind genaue Schätzungen der Ausführungszeiten von Anwendungsfunktionen und -aktivitäten für die jeweilige Zielplattform notwendig. Selbst wenn solche Schätzungen für bestimmte Plattformen durch aufwendige Analysen bereitgestellt werden können, so sind diese für andere Umgebungen nicht ohne weiteres übertragbar. Dies stellt eine Einschränkung des Freiheitsgrades für dynamische Umverteilungen und Anwendungsmigration auf beliebige Endgeräte dar.

Betrachtet man ferner die Ausführungszeiten von Aktivitäten innerhalb einer bekannten Plattform, so können diese in Wirklichkeit zur Laufzeit mehr oder weniger stark von den ursprünglichen Annahmen abweichen und schwanken. Gründe hierfür können unterschiedlicher Natur sein. Selbstverständlich hängt die effektive Ausführungszeit einer Funktion bis zur Bereitstellung eines Ergebnisses von der aktuellen Auslastung und die Synchronisation der Zugriffe auf gemeinsame Ressourcen innerhalb eines Systems ab. Hierbei ist vor allem der Grad der aktuellen CPU-Auslastung innerhalb einer Ausführungsplattform sowie der verwendete Rechenzeit-Zuteilungsalgorithmus für Aktivitäten entscheidend für die Einhaltung ihrer zeitlichen Fristen. Letzterer ist in der Regel abhängig von der verwendeten Softwareumgebung und dem Betriebssystem, welches den eigentlichen Zuteilungsmechanismus (Scheduler) implementiert. Ein weiterer Faktor sind Schwankungen der „reinen“ Ausführungszeiten, d.h. der tatsächlich benötigten Rechenzeit einer Aktivität auf einer Recheneinheit. Diese werden durch die Beschaffenheit und den Zustand der zugrundeliegenden Hardware-Plattform bedingt und sind nur schwer abschätzbar [10]. Gründe für solche Schwankungen sind sogar auf der Ebene einzelner Instruktionsanweisungen zu finden. So kann die benötigte Zeit für eine einzelne Instruktion um das Millionenfache ansteigen, abhängig vom Zustand der jeweiligen ausführenden Recheneinheit [11]. Hierbei spielen Faktoren wie die aktuelle Belegung von Prozessorcaches, Pipelines und der involvierten Translation Lookaside Buffers (TLB), sowie die variierenden Zugriffszeiten auf bestimmte RAM-Speicheradressen eine entscheidende Rolle. Letztere sind zum Beispiel für Architekturen mit mehreren Recheneinheiten und *Non-Uniform Memory Access* (NUMA) nicht deterministisch und hängen vom Speicherzugriffsmuster und dem aktuellen Zustand der einzelnen Recheneinheiten ab.

Die Folgen von Fehlschätzungen der Ausführungszeiten von Aktivitäten sind entweder eine Unter- oder eine Überprovisionierung der von ihnen benötigten Ressourcen. Wird die Rechenzeitanforderung einer Aktivität überprovisioniert, so stehen aus der Sicht eines Schedulers weniger Ressourcen für andere Aktivitäten bereit und die Rechenkapazität einer Plattform wird nicht optimal ausgeschöpft. Werden hingegen die Ressourcen unterprovisioniert, so können Aktivitäten nicht genügend Rechenzeit bekommen und ihre Fristen für die Bereitstellung von Ergebnissen verpassen. Letztere können sich als Fehler in das Gesamtsystem und in allen bereitgestellten Anwendungen ausbreiten, da eine Aktivität, welche mehr Rechenzeit benötigt als geplant, eventuell Ressourcen ausschöpft, die für andere Aktivitäten vorgesehen sind [12].

Im Forschungsfeld der Echtzeitsysteme liegt ein Schwerpunkt auf der Analyse der schlimmsten möglichen Ausführungszeit einer Aktivität (Task), der sog. Worst-Case Execution Time (WCET). Hierbei findet im Groben eine Analyse aller möglichen Ausführungspfade im Programm-Code einer Anwendung statt, unter Betrachtung der genauen Eigenschaften der un-

terliegenden Software- und Hardware-Architektur [13]. Das Ergebnis ist eine obere Schranke für die reine Rechenzeit einer jeden Aktivität, die dann von einem entsprechenden Zuteilungsalgorithmus verwendet wird. Aufgrund des nicht-deterministischen Verhaltens der Hardware ist allerdings eine exakte Abschätzung des „schlimmsten Falls“ nur bedingt möglich [13]. Die Analyse der WCET spielt eine entscheidende Rolle für harte Echtzeitsysteme. Dies sind zeit-kritische, meist reaktive Systeme (Flugzeuge, Kernkraftwerke, etc.), bei denen das Verpassen der Frist einer Aktivität (oder Task) dramatische Folgen haben könnte und deswegen nicht vorkommen darf. Für solche Systeme muss im Voraus sichergestellt werden, dass jede Aktivität genügend Rechenzeit bekommt, um in jedem Fall (auch im schlimmsten) ihre Frist zu schaffen. Da der schlimmste Fall in der Regel nur relativ selten eintritt, sind harte Echtzeitsysteme grundsätzlich durch eine durchschnittlich niedrige Ressourcenauslastung gekennzeichnet. Üblicherweise sind dynamische Umkonfigurationen der Task-Menge in solchen Systemen auch nicht zulässig, da diese für Nicht-Determinismus sorgen.

Bei den sogenannten „weichen“ Echtzeitsystemen werden Fristverletzungen grundsätzlich toleriert. Treten diese jedoch zu oft auf, so nimmt der Nutzen der bereitgestellten Ergebnisse und somit des Gesamtsystems ab. Wie dieser Nutzen und seine Toleranzgrenzen tatsächlich definiert sind, hängt vom jeweiligen Anwendungsszenario ab. In weichen Echtzeitsystemen werden in der Regel Schätzungen der mittleren Ausführungszeiten von Aktivitäten verwendet, den sog. Average-Case Execution Times (ACET). Diese Systeme erlauben grundsätzlich eine höhere Ressourcenauslastung, da hier keine Ressourcen für den schlimmsten Fall bereit gestellt werden müssen. Folglich können potentiell mehr Aktivitäten innerhalb einer Plattform bereitgestellt werden – man spricht hier von einer höheren Admissionsrate. Kommt es dabei zu Engpässen und Fristverletzungen, so existiert momentan kein standardisiertes Modell, wie sich ein System und die darin ausgeführten Anwendungen von Fehlern erholen können. Verschiedene Ansätze schlagen zum Beispiel Mechanismen zur dynamischen Admissionskontrolle vor, d.h. eine Kontrolle über die aktuell zugelassenen Aktivitäten, welche in einer Plattform ausgeführt werden dürfen. Andere Ansätze involvieren eine Reduzierung der von den Anwendungsaktivitäten generierten Last (Degradation), wobei entweder Arbeit gezielt ausgelassen wird oder die Komplexität der Anwendungsberechnungen auf funktionaler Ebene dynamisch reduziert wird. Entscheidend dabei bleibt die „Güte“ mit der das Gesamtsystem seine Aufgabe verrichtet. Einige solche Ansätze werden in Kapitel 2 vorgestellt und diskutiert.

Dynamische Rekonfigurationen der aktiven Menge von Anwendungen und Aktivitäten sind in weichen Echtzeitsystemen grundsätzlich legitim. Diese erfordern jedoch Mechanismen für den Umgang mit Engpässen, welche aufgrund von Überbelegung des Systems mit Aktivitäten oder ihrem schwankenden Verhalten zur Laufzeit entstehen können. Sind jedoch die Anwendungen

für die jeweilige Zielplattform „unbekannt“, d.h. es liegen keine (mittleren) Schätzungen für die Ausführungszeiten ihrer Aktivitäten vor oder diese sind nicht übertragbar, so müssen diese zur Laufzeit ermittelt werden. Eine dynamische Ressourcensteuerung ist dann notwendig, die mit Veränderungen der zugelassenen Menge von Tasks und ihrer Kostenschätzungen umgehen kann.

1.1 Problembeschreibung

Im Fokus dieser Arbeit liegt die Beurteilung bereits existierender Mechanismen und die Entwicklung eines neuen zentralen Ressourcenverwaltungssystems ARTOS, mit dem eine dynamische Menge von unbekannten Anwendungen und Aktivitäten mit weichen Echtzeitanforderungen innerhalb einer Plattform bewältigt werden kann. Die zentrale Ressource, die hier im Fokus liegt, ist die Rechenzeit, welche von den Anwendungen in Anspruch genommen wird. Neben der Admissionskontrolle und dem Umgang mit den unbekannten und zur Laufzeit schwankenden Ressourcenanforderungen der Anwendungen, soll ARTOS Strategien zur zentral gesteuerten Bewältigung von Engpässen bereitstellen. Dabei soll stets der bestmögliche Nutzen bzw. Qualität im System generiert werden, unter optimaler Ausschöpfung der vorhandenen Rechenkapazität der involvierten Hardware-Plattform. Darüber hinaus soll die Ressourcensteuerung eine möglichst fehlerfreie Ausführung der Anwendungsaktivitäten im Sinne ihrer Zeitschranken garantieren. Für Letzteres ist eine dynamische Analyse der Planbarkeit der Anwendungsaktivitäten notwendig, unter Betrachtung ihrer zur Laufzeit ermittelten Ausführungszeiten und Kosten. Diese „online“ Planbarkeitsanalyse stellt insofern eine Herausforderung dar, weil sie mit einem zusätzlichen Rechenaufwand und Komplexität verbunden ist, welcher im System ebenfalls eingeplant werden muss. Die Hauptaufgabe der dynamischen Ressourcensteuerung ist, einen gültigen Aktivierungsplan für die Aktivitäten innerhalb eines Systems zu erzeugen, welcher ihre aktuellen Rechenzeitanforderungen im Sinne ihrer Fristen berücksichtigt. Einer der Hauptbeiträge dieser Arbeit ist ein Modell, mit dem der Aufwand für die dynamische Erstellung eines solchen Plans und seine Verifikation anhand von Verhaltensprognosen der laufenden Anwendungen minimiert und zeitlich verteilt werden kann. Treten im Ressourcenverbrauch der Anwendungsaktivitäten zyklische Schwankungen auf, z.B. bedingt durch zyklisch auftretende Ereignisse, so soll die Steuerung diese ebenfalls detektieren und soweit wie möglich „maskieren“, damit der Aufwand für Entscheidungen über Nutzen und Anpassungen der Ressourcenverteilung bzw. der Anwendungs-konfiguration möglichst minimiert werden kann. Für Letzteres ist ein Mechanismus notwendig, welcher Ressourcen, die durch potentielle Überprovisionierung von Anwendungsanforderungen entstehen, dynamisch an aktuellen Verbrauchs-Hot-Spots verteilt.

Im Folgenden werden zunächst anhand eines Beispielszenarios die konkreten Anforderungen an ein ARTOS-System gestellt.

1.2 Beispiel-Szenario

Ein typisches Szenario für weiche Echtzeitsysteme sind Multimedia-Anwendungen, wie z.B. die Wiedergabe von Audio/Video-Inhalten (DVD, CD, MP3) oder Multimedia-Strömen (Video-on-Demand, Internettelefonie, Videokonferenzen). Solche Anwendungen sind gekennzeichnet durch zeitliche Fristen für ihre internen periodischen Aktivitäten, welche direkt aus den entsprechenden Anwendungsvorgaben stammen. Zum Beispiel muss ein Video mit einer bestimmten Bildrate wiedergegeben werden, um von einem Benutzer als fließend empfunden zu werden (ca. 25 Bilder pro Sekunde). Dabei müssen die bereitstehenden oder empfangenen Bilddaten innerhalb einer bestimmten Frist (bei 25 Bilder pro Sekunde = 40 Millisekunden) decodiert und auf einem Endgerät visualisiert werden. Gleichmaßen muss ein Audio-Decoder mit einer bestimmten Periode Audio-Daten verarbeiten und in die Ausgangspuffer einer Sound-Karte speichern, um ein kontinuierliches Ausgangssignal zu erzeugen. Wie bereits erwähnt, liegt der Fokus dieser Arbeit auf der von der Anwendung benötigten Rechenzeit im Endsystem und nicht auf der notwendigen Übertragungszeit für die Daten. Im Falle von Video-on-Demand oder Internettelefonie ist letztere abhängig von den Eigenschaften des zugrunde liegenden Kommunikationsnetzwerks, zum Beispiel von Faktoren wie seine Übertragungs-Bandbreite und -Latenz.

Die Qualität eines wiedergegebenen Videos hängt von format-spezifischen Faktoren ab, wie z.B. Bildgröße, Auflösung, Farbtiefe und weitere Codierungsfaktoren (z.B. Quantisierungsfaktor). Eine bessere Qualität ist in der Regel mit einem höheren Ressourcenaufwand innerhalb eines Video-Encoders bzw. Decoders verbunden. Eine spezielle Abhandlung möglicher Qualitätseinstellungen eines Videos und ihrer Auswirkungen wird später im Kapitel 2.4 vorgestellt. Somit ist der Rechenaufwand für die Darstellung eines jeden Bildes abhängig vom Format des Video-Signals. Ferner kann der Aufwand abhängig von den zu verarbeitenden Daten schwanken, z.B. kann er bei starker Bewegung innerhalb der wiedergegebenen Szene ansteigen, da nun Informationen von mehreren aufeinander folgenden Bildern für die aktuelle Darstellung relevant sind und verarbeitet werden müssen. Kann die Frist für die Darstellung eines Bildes nicht eingehalten werden, z.B. weil die entsprechende Aktivität nicht genügend Rechenzeit bekommt, so können Artefakte entstehen (Standbild, unvollständige Bilder, Verzerrungen), welche die subjektive Qualitätswahrnehmung eines Nutzers einschränken können. Nichtsdestotrotz, der Video-Strom wird nicht unterbrochen und ohne eine entsprechende Qualitätsanpassung würde

die Anwendung weiterhin mit der ihr verbleibenden Ressourcenmenge auskommen müssen und unter Umständen weitere Darstellungsfehler erzeugen.

Betrachtet man nur die Dekoder-Seite auf einem Endgerät, so kann eine Qualitätsanpassung zur Verringerung des Rechenaufwands mit oder ohne Veränderung des Videoformats stattfinden. So kann z.B. ein Empfänger in einem Streaming-Szenario den Sender anweisen, die Bild-Auflösung eines Videos herunter zu skalieren, die Bildrate abzusenken oder die Qualität der Einzelbilder zu reduzieren. Ebenso kann ein völlig anderes Video-Format im Sinne des verwendeten Codecs angefordert werden, welches weniger Ressourcen bei der Decodierung benötigt. Hat die Anwendung keine Kontrolle über das Format des Eingangssignals, so könnte sie einzelne Bilder (Reduzierung der Bildrate), oder bestimmte Informationen (z.B. Farbinformationen oder Enhancement-Layers im Falle eines MPEG-codierten Videos) verwerfen, um ihren Rechenaufwand zu verringern bis ein Engpass überwunden ist. Im Normalfall hat eine Anwendung jedoch weder Informationen über die Gründe oder das Auftreten von Engpässen, noch über den Erfolg der von ihr angewandten Anpassung und kann nur rückwirkend und optimistisch darauf schliessen. Hier kommt die zentrale Ressourcensteuerung ins Spiel, welche den Verbrauch der Anwendungen überwacht und globale Entscheidungen über Qualitätsanpassungen und über die aktuelle Ressourcenverteilung zwischen den Anwendungen treffen kann.

Betrachten wir nun folgendes Szenario wie dargestellt in Abbildung 1.1. Ein Benutzer spielt einen hoch-qualitativen Video-Stream über das Internet auf einem ressourcen-eingeschränkten Endgerät ab, welcher von einem VoD-Server übertragen wird (Abbildung 1.1a). Zu einem beliebigen Zeitpunkt kommt ein eingehender Videotelefonie-Anruf herein (Abbildung 1.1b). Das Ressourcenmanagementsystem stellt im Hintergrund einen Ressourcenengpass fest, da beide Anwendungen nicht mit ihrer aktuellen Qualität parallel ausgeführt werden können und fordert den Benutzer auf, eine der Anwendungen zu priorisieren (Abbildung 1.1c). Der Benutzer priorisiert den eingehenden VoIP-Anruf höher und die Qualität der VoD-Anwendung wird temporär heruntergesetzt, um den Engpass zu überwinden (Abbildung 1.1d).

1.3 Anforderungen

Aus dem vorherigen Beispiel-Szenario lassen sich unmittelbar einige Annahmen über die geforderte Ausführungsumgebung ableiten und Anforderungen an das zu entwickelnde Ressourcenverwaltungssystem von ARTOS definieren:

- a) *Dynamische Applikationsmenge*: Innerhalb einer Ausführungsplattform können Anwendungen dynamisch installiert, gestartet, gestoppt und deinstalliert werden.

Abbildung 1.1: Beispielszenario

- b) *Komponentenbasiertes und dienstorientiertes System:* Anwendungen können ihre mitgelieferten Code-Bibliotheken mit anderen Anwendungen auf der Basis von Komponenten-Definitionen teilen. Sie werden dynamisch zur Laufzeit aus Diensten anderer Anwendungen zusammengesetzt.
- c) *Variierende Ausführungszeiten ohne bekannte Obergrenze:* Die realen Ausführungszeiten von Anwendungsaktivitäten sind zunächst unbekannt und sie können zur Laufzeit stark schwanken.
- d) *Qualitätseinstellungen:* Die Qualität der Anwendungen und damit auch ihr Ressourcenverbrauch kann dynamisch eingestellt werden.
- e) *Benutzerpräferenzen:* Ein Benutzer oder das System selbst ist in der Lage Anwendungen zu priorisieren.

- f) *Seltene (defensive) Qualitätsanpassungen*: Vermeidung von Qualitätsanpassungen der Anwendungen, da diese selbst Kosten verursachen.
- g) *Planbarkeit der Applikationsaktivitäten*: Jede Anwendung soll genügend Ressourcen bekommen, so dass ihre interne Aktivitäten ihre zeitlichen Fristen schaffen.

Auf die einzelnen Punkte wird im Folgenden detailliert eingegangen:

Abbildung 1.2: Dynamische Applikationsmenge

Punkt (a) beschreibt ein weiches Echtzeitsystem in dem, wie bereits erklärt, Applikationen innerhalb einer Ausführungsplattform dynamisch hinzugefügt bzw. entfernt werden können (siehe Abbildung 1.2). Auf diese Weise verändert sich zur Laufzeit auch die Menge und die Beschaffenheit der im System zugelassenen Aktivitäten. Aus diesem Grund werden Mechanismen für dynamische Admissions- und Überlastkontrolle benötigt. Hier wird von einem System ausgegangen, bei dem Anwendungen temporär deaktiviert werden können, um eine optimale Verteilung der Ressourcen zu ermöglichen. Die dynamische Ressourcenverteilung wird bei ARTOS durch ein hierarchisches Zuteilungsmodell (Scheduling-Modell) realisiert, welches in Kapitel 3.2 weiter ausgeführt wird.

Abbildung 1.3: Komponentenbasierte Applikationen

Punkt (b) repräsentiert ähnliche Voraussetzungen, wie sie bereits für Anwendungen in modernen Betriebssystemen vorherrschen. Dort können Anwendungen ihren aktuellen Funktionsumfang durch das dynamische Laden und Einbinden von Programm-Code auf der Basis von Dynamic Shared Objects erweitern, welche unter Umständen von anderen Anwendungen bereitgestellt wurden. Der komponentenbasierte Ansatz, bei dem eine Anwendung aus Teilen anderer Applikationen zusammengesetzt wird (siehe Abbildung 1.3), wurde hier gewählt, um eine bessere und feingranulare Kontrolle über Versionierung, Aktualisierung und das Einbinden der geteilten Inhalte zu ermöglichen. Eine dienstorientierte Architektur (Service-Oriented Architecture (SOA)) ermöglicht wiederum eine feingranulare Auswahl und Komposition von Anwendungsfunktionen auf der Basis von Meta-Informationen und ist somit eine Abstraktionsstufe höher anzusiedeln als Komponentendefinitionen. Damit wird die dynamische Dienstausswahl bereits auf programmiersprachlicher Ebene durch Schnittstellendefinitionen der Dienste und einer zentralen Dienstregistratur auf Systemebene unterstützt (siehe Abbildung 1.4 – eine Aktivität τ innerhalb Anwendung A_3 ruft die Implementierung eines Dienstes S bereitgestellt durch Anwendung A_2 über eine zentrale Registry auf). Neue Anwendungskomponenten können somit neue Dienst-Implementierungen mitliefern, die ein spezialisiertes Verhalten einzelner Funktionen bereitstellen. Dieses Modell wurde bereits in ähnlicher Weise von Betriebssystemen wie Windows NT und Komponenten-Frameworks wie (D)COM, Corba und OSGi unterstützt. Die Meta-Informationen können auch mit Laufzeiteigenschaften der Dienstimplementierungen angereichert werden, zum Beispiel mit Werten von bereits für eine Ausführungsplattform ermittelten Ausführungszeiten einzelner Funktionen und die von ihnen erbrachte Qualität.

Abbildung 1.4: Dienstbasierte Applikationen

Die dynamische Auswahl von Dienstimplementierungen und deren Komposition zu Anwendungen erschwert die Ermittlung der tatsächlichen Ausführungszeiten von Aktivitäten, da diese nun vom aktuellen Bestand an Dienstimplementierungen innerhalb einer konkreten Plattform abhängt, welcher dynamisch variieren kann. Auf der anderen Seite ermöglicht die Dienstabstraktion eine einfache Realisierung von Qualitätsanpassungen, da nun dynamisch alternative und weniger rechen-aufwändige Implementierungen einzelner Funktionen mit unterschiedlicher erbrachter Qualität (z.B. Präzision einer Berechnung) ausgewählt werden können.

Da die realen Kosten der Aktivitäten sich erst zur Laufzeit innerhalb einer bestimmten Plattform etablieren und auch schwanken können, impliziert Punkt (c) zuverlässige Online-Messungen und Prognosen des Verhaltens einzelner Anwendungsaktivitäten. Die Realisierung solcher Messungen und der Prognose innerhalb des entwickelten ARTOS-Systems wird weiterhin im Kapitel 3.1 diskutiert.

Um eine gezielte Einstellung der Qualität einer Anwendung und somit ihres Ressourcenverbrauchs innerhalb einer Plattform zu ermöglichen (Punkt (d)), werden Anwendungen mit verschiedenen Betriebsmodi (oder Qualitätsstufen) definiert. Diese werden durch ein geeignetes Applikationsmodell unterstützt, welches in Kapitel 3.1 vorgestellt und weiter erläutert wird. Jeder Applikationsmodus generiert somit einen bestimmten und gezielt einstellbaren Nutzen (bzw. Qualität) unter Voraussetzung einer bestimmten Ressourcenmenge. Diese Ressourcenmenge muss innerhalb einer konkreten Ausführungsplattform durch online Messungen dynamisch ermittelt werden. Die Ressourcenverteilung zwischen den Anwendungen kann dann durch eine optimale Konfiguration von Anwendungsmodi realisiert werden. Generell ist anzunehmen, dass eine monotone Beziehung zwischen der Qualität und dem Ressourcenverbrauch von Anwendungen existiert – höhere Qualität benötigt mehr Ressourcen. Diese ist jedoch im entwickelten Modell in Kapitel 3.1 nicht zwingend. Die Auswahl der optimalen Konfiguration der Modi wird auf ein Optimierungsproblem zurückgeführt, welches zur Laufzeit gelöst wird. Das Ziel der Optimierung ist, den durch die aktuelle Auswahl von aktiven Anwendungsmodi erzielten Gesamtnutzen im System zu maximieren. Hier gibt es typischerweise einen Zielkonflikt zwischen der maximal einstellbaren Qualität und der optimalen Ressourcenauslastung innerhalb einer Plattform, bei schlechter Vorhersagbarkeit des Ressourcenverbrauchs der Anwendungen. Das Optimierungsproblem und der dafür gewählte Lösungsansatz werden in Kapitel 3.2 näher erläutert.

In Punkt (e) werden Anwendungsprioritäten beschrieben, welche anhand von Benutzerpräferenzen über bestimmte Applikationen eingestellt werden. Dabei sollen höher priorisierte Anwendungen potentiell in höhere Modi aktiviert werden. Die letztendliche Konfiguration der Modi wird jedoch durch den Optimierungsalgorithmus entschieden (siehe Punkt (d)), welcher die Prioritäten der Anwendungen mit dem generierten Nutzen (oder Qualität) und den Ressourcenverbrauch ihrer unterschiedlichen Betriebsmodi abwägt.

In Punkt (f) wird davon ausgegangen, dass die Umschaltung einer Anwendung in unterschiedlichen Modi mit zusätzlichen Kosten verbunden ist. Diese Kosten können durch modus-spezifische Anpassungen von Berechnungsdaten, Rekombination der Anwendungslogik oder durch Anpassung der Laufzeiteigenschaften von Anwendungsaktivitäten (siehe Kapitel 3.1) bedingt sein. Aus diesem Grund sollen Rekonfigurationen möglichst selten oder „defensiv“ betrieben werden.

Problematisch sind hierbei zyklische Schwankungen in den Verbräuchen von Anwendungen. Diese können durch zyklisch auftretende Ereignisse oder Inhalte innerhalb der Berechnungsdaten (z.B. I-Frames innerhalb eines Video-Signals, siehe Kapitel 2.3) hervorgerufen werden. Es sind also Mechanismen zur Glättung zyklisch auftretender Bursts und zur Vermeidung von Rekonfigurationszyklen notwendig. Diese können zum Beispiel durch eine feingranulare Verteilung von freien Kapazitäten an Verbrauchshotspots innerhalb einer Plattform erreicht werden. Die genaue Realisierung solcher Mechanismen wird in Kapitel 3.2 diskutiert.

Im Punkt (g) wird gefordert, dass die Anwendungsaktivitäten bezüglich ihren zeitlichen Fristen stets planbar bleiben. Da die realen Kosten der Aktivitäten sich erst zur Laufzeit innerhalb einer bestimmten Plattform etablieren und auch schwanken können, sind online Planbarkeitstests notwendig. Dies ist ein schwieriges Unterfangen, weil es zunächst nicht klar ist, wann und wie oft Planbarkeitstests durchgeführt werden müssen. Die Tests müssen jedoch die Gesamtmenge von Aktivitäten innerhalb einer Ausführungsplattform und auch die Konfigurationsmöglichkeiten der Anwendungen in unterschiedlichen Qualitätsstufen berücksichtigen. Im Kapitel 3.2 wird dafür ein Ansatz vorgestellt, welcher den Aufwand für solche Planbarkeitstests deutlich reduziert und zeitlich verteilt. Diese werden lokal nur für bestimmte Applikationen betrieben, für die die Prognose des Task-Verhaltens (Punkt (c)) nicht mehr zutreffend ist. Die Planbarkeitstests sind ferner mit der Optimierung der Anwendungsmodi gekoppelt, so dass eine falsche Prognose eine Rekonfiguration des Gesamtsystems auslösen kann.

Anhand dieser Anforderungen können nun bereits existierende Ansätze und Mechanismen für dynamische Ressourcensteuerungen innerhalb weicher Echtzeitsysteme identifiziert, gegenüber gestellt und bewertet werden. Kapitel 2.1 behandelt einige theoretische Grundlagen aus dem Bereich der Echtzeitsysteme, welche für das Verständnis der in Kapitel 2.2 präsentierten Ansätze notwendig sind. Zunächst wird jedoch in Kapitel 1.4 die Arbeit thematisch eingeordnet.

1.4 Thematische Einordnung der Arbeit

Das in dieser Arbeit konzipierte Ressourcensteuerungssystem kann zum Beispiel innerhalb eines Betriebssystems (Abbildung 1.5a) oder auf einer höheren Ebene, z.B. als Teil von einer Middleware (Abbildung 1.5b), integriert werden. Um ein Maß für die Güte der Ressourcensteuerung zu erlangen, impliziert das Modell Echtzeitanforderungen von Anwendungsaktivitäten und vergibt Echtzeitgarantien für diese Aktivitäten, welche durch entsprechende adaptive Ressourcenreservierungen erfüllt werden sollen. Somit beschreibt die Menge von Verletzungen dieser Garantien die Güte der Steuerung. Ein interessanter Aspekt dabei ist, wie gut ein solches System an die

Abbildung 1.5: Integration des Ressourcenmanagement-Frameworks

Erfüllung von harten Echtzeitgarantien herankommt, d.h. inwiefern Fristverletzungen durch die vorgestellten Ansätze und Mechanismen grundsätzlich verhindert werden können. Dieser Aspekt wird abschließend im Kapitel 5 diskutiert.

Diese Arbeit definiert sich thematisch interdisziplinär über die Themenbereiche *Systemnahe Programmierung* und *Echtzeitsysteme* – die Arbeit ist somit eine Synthese aus Systemtheorie und Echtzeittheorie (siehe Abbildung 1.6). Die Echtzeittheorie wird also lediglich verwendet, um Anwendungsanforderungen zu erfassen, Ressourcen dynamisch zu planen und Garantien zu vergeben sowie auf Verletzungen von Anforderungen zu reagieren.

Abbildung 1.6: Thematische Einordnung der Arbeit

1.5 Wissenschaftliche Beiträge

Im Rahmen dieser Arbeit wurde ein offenes Echtzeit-System ARTOS entwickelt, das mit einer dynamischen Menge von Anwendungen mit unbekannten und schwankenden Ressourcenverbräuchen umgehen kann. Hierfür werden die Laufzeitkosten der Anwendungen online approximiert und ihre Tasks innerhalb eines dreistufigen hierarchischen Scheduling-Modells integriert.

Auf der höchsten Ebene des Scheduling Modells wurde ein Optimierungsverfahren entwickelt, das in einer pseudo-polynomiell begrenzten Zeit eine optimale Konfiguration der Anwendungen in unterschiedlichen Betriebsmodi berechnet und aktiviert. Eine ausführliche Evaluation der Optimierung wird im Rahmen der Arbeit vorgestellt. Die Ergebnisse wurden bereits auch als technischer Bericht in [14] publiziert.

Für die Berechnung der optimalen Konfiguration verwendet die Optimierung online geschätzte und adaptive Ressourcenbandbreiten der Anwendungsmodi. Diese Bandbreiten dienen auch als Grundlage für die von ARTOS gesteuerte Aktivierung der Anwendungen, wofür spezielle Constant Bandwidth Server (CBS)-Mechanismen und ein eigens entwickeltes prioritäts-gesteuertes Scheduling entwickelt wurden. Dabei wurde eine spezielle Lösung entwickelt, bei der ein CBS gleichzeitig mehrere Anwendungs-Tasks unterstützt und somit ganze Anwendungen kapseln kann. Die bisher existierenden Arbeiten im Kontext von CBS sehen nur einzelne Tasks innerhalb der Server vor, was generell zu einem höheren Scheduling-Overhead als beim hier vorgestellten Modell sorgt. Die Arbeit zeigt vor allem auch, wie die entsprechende Kapazität und Periode der CBS so eingestellt werden können, dass die anwendungs-lokalen Echtzeit-Anforderungen der Tasks nicht durch das Scheduling der Server verletzt werden. Die entwickelten Mechanismen wurden bereits im Rahmen einer wissenschaftlichen Zeitschrift [15] veröffentlicht und auf der gemeinsamen Tagung der GI-Fachausschüsse Echtzeitsysteme und Betriebssysteme in Boppard am Rhein vorgestellt. Sie wurden dort mit einem „Best Paper Award“ ausgezeichnet.

Auf der untersten Task-Ebene realisiert das Modell ein benutzerspezifisches bzw. applikations-lokales Scheduling. Dieses findet innerhalb der CBS der Anwendungen statt, also im Zusammenspiel mit dem Scheduling auf der Applikations-Ebene. Solche hierarchische Scheduling-Modelle existieren zwar bereits (z.B. im MaRTE OS [16]), jedoch nicht im Zusammenhang mit dynamisch begrenzten Ressourcenkapazitäten und Echtzeit-Anforderungen der Anwendungen. Letztere verlangen eine Abbildung der Kapazitäts-Grenzen der Anwendungen auf das Scheduling der einzelnen Tasks, was im Rahmen dieser Arbeit durch die angepassten Funktionen der CBS ermöglicht wird.

Im Rahmen dieser Arbeit werden unterschiedliche Möglichkeiten zur Approximation der Task-Kosten zur Laufzeit bewertet und ein entsprechendes Modell vorgeschlagen. Dieses Modell ermöglicht eine zeitweilige Approximation der Rechenzeitanforderungen der Tasks mit einem dynamisch anpassbaren Fehler. Dieser Fehler etabliert sich als eine Überprovisionierung der Tasks, abhängig von ihrem aktuellen Schwankungs-Verhalten. Die Approximationen ermöglichen eine Verzögerung der Scheduling-Aktivitäten innerhalb der höheren Ebenen des Modells, welche in einem laufenden System mit Kosten verbunden sind. Auf dieser Basis schlägt ARTOS ein Modell vor, bei dem die Kosten für die Online-Analyse der Planbarkeit der Anwendungen und für die Optimierung ihrer Konfiguration minimiert werden können. Diese Modell wurde im Rahmen eines Beitrags [17] auf der RTAS'15-Konferenz in Seattle vorgestellt.

Die vorgeschlagenen Mechanismen wurden mit der Real-Time Specification for Java (RTSJ) implementiert und innerhalb eines komponenten-basierten Frameworks integriert, welches als Prototyp eines ARTOS-Systems dient. Anhand dieser Implementierung konnte die Wirksamkeit des Modells evaluiert und bewiesen werden. Da sich allerdings RTSJ nur bedingt als Implementierungs-Grundlage geeignet erwiesen hat, wurden entsprechende Änderungen der Spezifikation erarbeitet und zur Standardisierung vorgeschlagen. Zum Zeitpunkt der Bearbeitung dieser Promotionsarbeit wurde eine neue Version der Spezifikation erstellt, so dass die vorgeschlagenen Mechanismen direkt in die neuen Spezifikations-Drafts aufgenommen werden konnten. Die Diskussionen mit der RTSJ Expert Group fanden unter anderem im Rahmen der JTRES'15-Konferenz in Paris statt, wo die vorgeschlagenen RTSJ-Erweiterungen im Kontext des hierarchischen Scheduling-Modells publiziert und präsentiert wurden.

Das in dieser Arbeit entwickelte Modell und seine Mechanismen finden prinzipiell in allen Systemen eine Anwendung, bei denen Ressourcen dynamisch gesteuert und auch Echtzeitanforderungen von Anwendungen erfüllt werden müssen. Diese können zum Beispiel Cloud- oder HPC-Systeme innerhalb großer Datenzentren mit hunderten von vernetzten Prozessor-Einheiten sein, aber auch mobile und eingebettete Geräte mit beschränkten Ressourcen und Rechenkapazitäten. Der Einsatz von ARTOS im Rahmen dynamisch gesteuerter Cloud-Systeme wurde bereits beim 6th Cloud Control Workshop im Rahmen der UCC'14-Konferenz in London vorgestellt [1]. Eine Integration der Mechanismen auf der Basis der `SCHED_DEADLINE` Scheduling-Policy im Linux Kernel ist im Rahmen eines Forschungsprojekts zusammen mit der ReTiS-Gruppe der Scuola Superiore Sant'Anna und der Firma Aicas in Planung. Darüber hinaus soll ARTOS als Basis für ein Forschungsprojekt im Bereich von HPC-Systemen dienen und dort für eine Performanz-Steigerung durch dynamische Ressourcen-Steuerung dienen. Eine Integration des Modells innerhalb des MyThOS-Betriebssystems [18] soll eine Basisplattform für das geplante Forschungsprojekt bereitstellen. Durch den minimalen Overhead von MyThOS können dann auch eingebettete

Systeme als Ausführungs-Plattformen anvisiert werden, wo eine adaptive Ressourcen-Planung unter Beachtung von Echtzeitanforderungen von besonderer Bedeutung ist.

1.6 Überblick

Nach dieser Einleitung der Arbeit werden nun im Folgenden zunächst Grundlagen und der Stand der Forschung präsentiert (Kapitel 2). Dieser Abschnitt gliedert sich mit einer kurzen Einführung in die Echtzeittheorie (Kapitel 2.1), eine Abhandlung der verwandten Arbeiten (Kapitel 2.2), Grundlagen der Video-Codierung und Übertragung (Kapitel 2.3) sowie Mechanismen für Qualitätsanpassungen (Kapitel 2.4) in mehreren Unterabschnitten. Die im Kapitel 2.1 vorgestellten echtzeit-theoretischen Grundlagen werden nur soweit vertieft, wie es das Verständnis der in dieser Arbeit entwickelten Mechanismen abverlangt.

Auf der Basis der vorgestellten Grundlagen und Anforderungen wird im Kapitel 3.1 ein Applikationsmodell für ARTOS-Anwendungen und Aktivitäten mit mehreren Betriebsmodi definiert. Anhand dessen werden im Kapitel 3.2 die hierarchischen Scheduling-Mechanismen des Systems entworfen, deren prototypische Implementierung im ARTOS-Framework später in Kapitel 4 näher vorgestellt wird. Dort wird nach einer Vorstellung des verwendeten Java-Komponentenmodells (Kapitel 4.1), von RTSJ (Kapitel 4.2) und der gewählten Ausführungsplattform (Kapitel 4.3) separat auf die Implementierung der einzelnen Ebenen des Schedulers eingegangen 4.4 und eine Bewertung und Erweiterung der verwendeten RTSJ-Mechanismen betrieben 4.5.

Im Kapitel 5 werden die in dieser Arbeit entwickelten Mechanismen evaluiert und ihre Wirksamkeit getestet. Nach einer Beschreibung der Evaluationsumgebung (Kapitel 5.1) werden in den Kapiteln 5.2 und 5.3 die Scheduling-Mechanismen auf der Task- und Applikations-Ebene getestet. Im Kapitel 5.4 werden wiederum die automatischen Adaptionen-Mechanismen des Frameworks evaluiert, während im Kapitel 5.5 die Erfassung periodischer Bursts durch das System auf der Task-Ebene diskutiert wird. Nach einer Evaluation der auf der System-Ebene des Modells angesiedelten Optimierung im Kapitel 5.6, wird im Kapitel 5.7 eine ARTOS-kompatible Video-on-Demand Anwendung entwickelt und für eine Evaluation des durch die automatischen Adaptionen-Mechanismen der Anwendungen in ARTOS gewonnenen Nutzens herangezogen.

Die Arbeit endet im Kapitel 6 mit einer Zusammenfassung ihrer Ergebnisse und Beiträge, sowie mit einem Ausblick über zukünftige Arbeiten und mögliche Erweiterungen der vorgestellten Mechanismen.

2

Grundlagen und Stand der Forschung

In diesem Kapitel werden zunächst die theoretischen Grundlagen von Echtzeitsystemen vorgestellt, welche anschließend als Basis für die Abgrenzung dieser Arbeit gegenüber bereits existierenden verwandten Ansätzen dienen. Dabei wird auf unterschiedliche Task-Modelle, spezielle Ablaufplanungs-Mechanismen und den Nachweis der Planbarkeit einer Task-Menge sowie die damit verbundene Komplexität eingegangen. Auf der Basis dieser Definitionen und Konzepte wird später im Kapitel 3 ein Anwendungsmodell für ARTOS-kompatible Applikationen mit mehreren Betriebsmodi geschaffen und der in dieser Arbeit zentrale Ansatz des hierarchischen Scheduling-Modells für unbekannte Anwendungen mit schwankenden Ressourcenanforderungen entwickelt. Nach einer Vorstellung der verwandten Arbeiten wird im weiteren Verlauf des Kapitels auf Realisierungsmöglichkeiten von adaptiven Anwendungen mit mehreren Qualitätsstufen eingegangen. Der Fokus liegt dabei auf multimedialen Applikationen wie z.B. die Übertragung und Wiedergabe von qualitativ adaptierbaren Video-Inhalten. Dafür werden im Kapitel 2.3 einige Grundlagen der Video-Codierung und -Übertragung vorgestellt. Dabei werden auch Gründe für potentielle Qualitätsanpassungen in entsprechenden Anwendungsszenarien diskutiert. Ausgehend davon werden dann im Kapitel 2.4 generelle Ansätze für Qualitätsanpassungen vorgestellt. Hierbei werden zunächst Konzepte für die Realisierung unterschiedlicher Anwendungsmodi diskutiert. Darauf aufbauend werden anhand von zwei Fallstudien, für Anpassungen der

Abbildung 2.1: Task-Modell

Qualität von Video-Signalen (Kapitel 2.4.2) und von dynamisch bereitgestellten Web-Inhalten (Kapitel 2.4.3), spezielle technische Mechanismen zur Steuerung des Ressourcenverbrauchs von adaptiven Anwendungen durch Veränderung ihrer Qualität vorgestellt und bewertet.

Dieses Kapitel dient als Grundlage für die weitere Vorstellung des entwickelten hierarchischen Scheduling-Modells. Im Folgenden wird auf dieser Grundlage immer wieder verwiesen.

2.1 Echtzeittheorie

In Echtzeitsystemen hängt die Korrektheit einer Berechnung grundsätzlich vom Ergebnis der Berechnung, aber auch vom Zeitpunkt seiner Auslieferung ab. Die zentrale Einheit, die eine solche Berechnung repräsentiert, ist eine Aufgabe oder *Task* τ . Aus der Anwendungssicht wird eine Task durch einen Aktivitätsträger innerhalb einer Zielplattform realisiert, der durch eine Menge von Anweisungen und Instruktionen für einen Prozessor definiert wird. In einem Echtzeitsystem muss die Ausführung des Aktivitätsträgers zeitlich so eingeplant werden, dass er „rechtzeitig“ fertig werden und ein Ergebnis seiner Berechnung innerhalb einer zeitlichen Frist ausliefern kann.

Abbildung 2.1 veranschaulicht die wichtigsten Parameter, welche bei der Ausführung einer Task eine Rolle spielen. So wird eine Task τ zu einem bestimmten Zeitpunkt r aktiviert (*Release Time*) und ihre Ausführung ist mit einer bestimmten Menge an Rechenzeit bzw. Ausführungskosten C (*Cost*) verbunden. Eine Task besitzt ebenfalls eine Frist D (*Deadline*), innerhalb welcher das Ergebnis ihrer Berechnung spätestens ausgeliefert werden muss. Diese Frist kann relativ zum Aktivierungszeitpunkt der Task oder absolut in der Zeit definiert sein. Kann die Frist nicht eingehalten werden, dann spricht man von einer sog. Fristverletzung (*Deadline Miss*).

Die Definition von Echtzeitsystemen spaltet sich grundsätzlich in sog. *harte* und *weiche* Echtzeitsysteme (siehe Abbildung 2.2). Bei der harten Echtzeit werden Fristverletzungen nicht toleriert. Ein Ergebnis, das nach seiner entsprechenden Frist ausgeliefert wird, hat keinen Nutzen U mehr

Abbildung 2.2: Harte und weiche Echtzeit

für die Anwendung. Harte Echtzeitsysteme rechnen mit den schlimmsten Ausführungszeiten von Tasks, den sog. *Worst-Case-Execution-Times* (WCET), da hier die erforderlichen Ressourcen für eine fehlerfreie Ausführung in jedem Fall bereitgestellt und garantiert werden müssen. Diese Anforderung wird bei weichen Echtzeitsystemen abgeschwächt, da hier Fristverletzungen toleriert werden können. Dabei fällt der Nutzen eines Ergebnisses nach seiner entsprechenden Frist mit der Zeit ab (vgl. Abbildung 2.2). Aus diesem Grund muss bei weichen Echtzeitsystemen nicht mit dem schlimmsten Fall gerechnet werden und es können mittlere Ausführungszeiten der Tasks, sog. *Average-Case-Execution-Times* (ACET), verwendet werden. Da harte Echtzeitsysteme die Ressourcen für den schlimmsten Fall garantieren müssen, dieser aber in der Regel nur selten eintritt, leiden sie generell unter einer geringen Auslastung und somit Ressourcenverschwendung. Weiche Echtzeitsysteme ermöglichen prinzipiell eine höhere Auslastung der verfügbaren Ressourcen, da hier grundsätzlich eine höhere Admissionsrate von Tasks erreicht werden kann. Sie benötigen jedoch Mechanismen zur Behandlung von Fristverletzungen und für den Umgang mit Überlastsituationen [12].

Es existieren unterschiedliche Task-Modelle, die sich in der Definition und in der Aktivierungsweise der Tasks unterscheiden. In dieser Arbeit wird zunächst nur auf *aperiodische* und *periodische* Tasks eingegangen; andere Task-Modelle, wie z.B. *sporadische* Tasks, können jedoch ebenso im vorgestellten hierarchischen Scheduling-Modell berücksichtigt werden. Aperiodische Tasks besitzen nur eine Ausführungsinstanz (*Job*), diese wird ein einziges Mal aktiviert und die Task bleibt solange aktiv, bis ihre Berechnung vollständig ausgeführt und das entsprechende Ergebnis ausgeliefert wurde (vgl. Abbildung 2.3a). Eine aperiodische Task wird somit definiert durch einen Aktivierungszeitpunkt r , ihre Kosten C und eine Frist D , also durch $\tau_{\text{aperiodic}} = (r, C, D)$. Periodische Tasks besitzen hingegen mehrere Ausführungsinstanzen (Jobs), die mit einer bestimmten Rate bzw. Periode T aktiviert werden (vgl. Abbildung 2.3b). Die Frist einer periodischen Task ist relativ zum Beginn ihrer aktuellen Periode und etabliert sich als absolute Frist für ihren jewei-

ligen Job. Die Definition einer periodischen Task wird entsprechend um die Periode T erweitert, also ist $\tau_{periodic} = (r, T, C, D)$. Werden alle Tasks in einem System mit einem absoluten Aktivierungszeitpunkt r versehen, so spricht man von einer *konkreten* Taskmenge; anderenfalls von einer *unkonkreten* Taskmenge. Eine Fristverletzung kann vorkommen, beispielsweise wenn sich die Kosten einer Task für einen oder mehrere Jobs erhöhen, oder wenn die Task nicht genügend Rechenzeit bekommt.

Abbildung 2.3: Aperiodische und periodische Task-Modelle

Die zentrale Ressource bei den Echtzeitsystemen ist die Rechenzeit (*CPU time*), neben anderen Ressourcen wie Speicher oder Netzwerk, die ebenfalls modelliert werden können. In dieser Arbeit liegt der Fokus nur auf der Rechenzeit, obwohl die Ergebnisse auch auf andere Ressourcen-Arten übertragen werden können. Die Vergabe von Rechenzeit an Tasks wird in einem Echtzeitsystem durch einen *Scheduler*¹ gesteuert. Dieser entscheidet darüber, welche Task wann aktiviert und auf einer Recheneinheit bzw. einem Prozessor ausgeführt wird. In dieser Arbeit wird zunächst von einem Prozessor ausgegangen, wobei Multiprozessor-Umgebungen ebenfalls in Betracht gezogen werden können. Die Aufgabe der Planung ist somit die Organisation der Tasks in einem sequentiellen Ablauf, wobei jede Task zweifachweise den Prozessor zur Ausführung zugewiesen bekommt. Grundsätzlich existieren *prioritäts-* und *zeitschränken-gesteuerte* Ablaufplanungsverfahren. In Kapitel 2.1.1 wird ein zeitschränken-gesteuertes Verfahren näher vorgestellt, welches auf mehreren Ebenen des in dieser Arbeit entwickelten hierarchischen Scheduling-Modells zum Einsatz kommt.

Eines der zentralen Probleme innerhalb Echtzeitsysteme ist die Planbarkeit (*Feasibility*) der involvierten Taskmenge, d.h. die Frage ob alle Tasks ihre Fristen einhalten werden. Da der Ablauf der Tasks abhängig vom entsprechenden Ablaufplanungsverfahren (*Scheduling Mechanism*) ist, unterliegt auch ein entsprechender Planbarkeitsnachweis (*Feasibility Test*) diesem Verfahren. Darüber hinaus ist die Planbarkeit einer Taskmenge durch die tatsächliche Ausprägung der Tasks, im Sinne ihrer Perioden, Kosten und Deadlines, bedingt.

Um die Problematik des Planbarkeitsnachweises näher zu erläutern, wird zunächst ein Ablauf-

¹In den gängigsten Konfigurationen ist der Scheduler Teil eines zugrundeliegenden Echtzeit-Betriebssystems.

planungsverfahren näher vorgestellt, welches innerhalb Echtzeitsystemen eine breite Anwendung findet und speziell in dieser Arbeit auf mehreren Ebenen des entwickelten hierarchischen Scheduling-Modells zum Einsatz kommt. In Kapitel 2.1.2 wird dann näher auf den Planbarkeitsnachweis am Beispiel dieses Verfahrens eingegangen.

2.1.1 Ablaufplanung

Earliest Deadline First (EDF) ist ein zeitschranken-gesteuertes Scheduling-Verfahren mit *dynamischen* Prioritäten. Dies bedeutet in erster Linie, dass ein konkreter Ablauf von Task-Aktivierungen hauptsächlich durch das Auftreten der Task-Fristen (bzw. Zeitschranken) bestimmt wird. Hierbei bekommt zu einem bestimmten Zeitpunkt die Task mit der *kleinsten* absoluten Frist die *höchste* Priorität. Wir gehen von einem *präemptiven* Task-Modell aus, bei dem eine Task mit einer höheren Priorität eine andere, aktuell den Prozessor belegende Task mit niedrigerer Priorität unterbrechen und verdrängen kann. EDF ist außerdem ein optimales Ablaufplanungsverfahren. D.h. wenn eine Taskmenge mit einem Algorithmus planbar ist, dann ist sie in jedem Fall auch mit EDF planbar [19]. Diese Optimalitätseigenschaft von EDF wird in der Theorie ausgenutzt, da es klar ist, dass wenn eine Task-Menge mit EDF nicht planbar ist, sie dann mit keinem anderen Verfahren geplant werden kann. Bei anderen nicht-optimalen Verfahren, wie z.B. das *Rate Monotonic Scheduling* (RMS) [20, 21], kann es dagegen vorkommen, dass eine Task-Menge als nicht-planbar eingestuft wird, obwohl sie in einem konkreten System fehlerfrei abläuft. Bei einem RMS-basierten System werden Tasks feste Prioritäten nach ihren Periodenlängen vergeben - Tasks mit kleineren Perioden bekommen dabei höhere Prioritäten als Tasks mit größeren Perioden. EDF ist auch gut geeignet für Systeme mit relativ hoher Auslastung, da es dann immer noch einen gültigen Plan erzeugt – anders als bei RMS-basierten Systemen, die bis zu einer CPU-Auslastung von 69,3 % planbar bleiben und danach nur heuristische Aussagen über ihre Planbarkeit getroffen werden können [22].

EDF gilt generell als schwierig zu implementieren, da zu jedem Zeitpunkt überprüft werden muss, welche Task die kleinste Frist besitzt. Lelli et al. haben bereits einen `SCHED_DEADLINE` genannten EDF-Scheduler [23, 24] für einen Linux-Kernel entwickelt. Dieser wurde im Rahmen des JUNIPER-Projekts [25] als Basis für einen hierarchischen Scheduler [26] für Multiprozessoren verwendet. Basierend auf `SCHED_DEADLINE` kann prinzipiell auch das in dieser Arbeit vorgestellte Modell realisiert werden. Da für die Implementierung unseres Modells ein Ansatz basierend auf Echtzeit-Java [27] gewählt wurde, wurden benutzerdefinierte EDF-Mechanismen entwickelt, welche im Kapitel 4 näher vorgestellt werden.

Abbildung 2.4: Ablaufplanung mit EDF

Des Weiteren wird eine Beispiel-Taskmenge Γ von zwei Taks $\tau = (T, C, D)$ mit $\tau_1 = (5, 2, 4)$ und $\tau_2 = (3, 1, 2)$ als laufendes Beispiel eingeführt auf das in den folgenden Kapiteln stets verwiesen wird. Die Aktivierung dieser Taksmenge mit EDF verhält sich wie in Abbildung 2.4.

Ausgegangen von einer synchronen² Task-Aktivierung zum Zeitpunkt t_0 , besitzt zunächst die Task τ_2 eine kürzere absolute Frist im Zeitpunkt $t = 2$ und der erste Job von τ_2 wird darum zuerst mit Kosten = 1 Zeiteinheiten (ZE) ausgeführt. Nach Beendigung des ersten Jobs von τ_2 wird zum Zeitpunkt $t = 1$ der erste Job von τ_1 mit Kosten = 2 ZE gestartet. Zum Zeitpunkt $t = 3$ beginnt die neue Periode von τ_2 und der zweite Job der Task wird instantan ausgeführt, da τ_1 die Ausführung ihres ersten Jobs bereits beendet hat und für sie keine neue Periode begonnen hat. Zum Zeitpunkt $t = 5$ startet die neue Periode und somit der zweite Job von τ_1 . Bei $t = 6$ beginnt jedoch eine neue Periode von τ_2 mit einer kürzeren Frist = 8 ZE als die der aktuell ausgeführten Task τ_1 . Dies führt zu einer sofortigen Unterbrechung von τ_1 , so dass der aktuelle Job von τ_2 zuerst ausgeführt wird, bevor die unterbrochene Task τ_1 fortgesetzt wird. Die Task-Aktivierung setzt sich dann analog fort.

Eine interessante Eigenschaft von EDF ist, dass die Aktivierungsreihenfolge der Tasks sich innerhalb ihrer *Hyperperiode* H wiederholt. Die Hyperperiode ist definiert als das *kleinste gemeinsame Vielfache* (KGV) der Task-Perioden (für das obere Beispiel ist $H = 15$ ZE). Diese Eigenschaft hat die Konsequenz, dass die Planbarkeit einer mit EDF gesteuerten Taskmenge, nur innerhalb der Hyperperiode der Tasks nachgewiesen werden muss [22]. Wie man in Abbildung 2.4 sieht, ist also die Taskmenge Γ planbar, da keine der Tasks eine Fristverletzung innerhalb der Hyperperiode H erleidet.

In einem realen Echtzeitsystem muss die Planbarkeit der involvierten Taskmenge entweder vor der eigentlichen Ausführung des Systems (*offline*), anhand im Voraus ermittelter Taskkosten-Schätzungen, oder zur Laufzeit (*online*), anhand aktuell gemessener Taskkosten, bestimmt wer-

²d.h. beide Tasks starten gleichzeitig zum Zeitpunkt t_0

den. Ein Planbarkeitsnachweis birgt jedoch eine gewisse Komplexität in sich, die in bestimmten Fällen einen hohen Overhead im System erzeugen kann – gerade wenn dieser online berechnet werden muss. Im nachfolgenden Kapitel 2.1.2 werden die Grundvoraussetzungen und -mechanismen für Planbarkeitsnachweise für Taskmengen, die mit EDF aktiviert werden, vorgestellt. Der Fokus liegt dabei auf der mit dem Nachweis verbundenen Komplexität und auf Algorithmen, die auf eine Verminderung dieser Komplexität abzielen.

2.1.2 Planbarkeitsnachweis

Die im Folgenden vorgestellten Mechanismen der Planbarkeitsanalyse beziehen sich zunächst auf eine periodische und mit EDF gesteuerte Taskmenge, so wie sie im Rahmen dieser Arbeit vorausgesetzt wurde.

Eine Taskmenge $\Gamma = \{\tau_i \mid i = 1, \dots, n\}$ ist planbar, wenn formal nachgewiesen werden kann, dass *alle* Tasks $\tau \in \Gamma$ ihre Fristen schaffen. Der damit verbundene formale Test heisst Planbarkeitsnachweis (*Feasibility Test*). Für die Betrachtung des Planbarkeitsnachweises ist zunächst die Definition einiger weiterer Begriffe notwendig.

Die **Auslastung** (*Utilization*), welche von einer Taskmenge erzeugt wird, definiert die Menge an Prozessorzeit, die bei der Ausführung der Tasks benötigt wird. Die Auslastung ist gegeben durch:

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \quad (2.1)$$

also durch die Summe der Quotienten aus Taskkosten und Taskperioden. Für die weiter oben im Kapitel 2.1.1 definierte Beispiel-Taskmenge von zwei Tasks $\tau = (T, C, D)$ mit $\tau_1 = (5, 2, 4)$ und $\tau_2 = (3, 1, 2)$ ergibt sich somit im System eine Auslastung von:

$$U = \frac{1}{3} + \frac{2}{5} = \frac{11}{15} = 0,73 \quad (\text{also } 73 \%)$$

Eine notwendige Bedingung für die Planbarkeit einer jeden periodischen Taskmenge ist, dass:

$$U \leq 1 \quad (2.2)$$

die Auslastung im System kleiner gleich 1 (also 100 %) ist.

Falls $\forall \tau_i \in \Gamma$ gilt $D_i = T_i$, also Task-Fristen sind gleich den Task-Perioden, dann ist $U \leq 1$ ebenfalls eine hinreichende Bedingung für die Planbarkeit der Tasks. Somit hätte der Planbarkeitsnachweis über den Auslastungstest eine lineare Komplexität von $\mathcal{O}(n)$. Haben jedoch Tasks

mit $D_i \leq T_i$ Fristen kleiner-gleich ihren Perioden, oder gar $D_i \geq T_i$, dann reicht der Test über die Auslastung des Prozessors nicht als Planbarkeitsnachweis aus und die Rechenzeitanforderung der Tasks muss in Betracht gezogen werden.

Die **Rechenzeitanforderung** (*Processor Demand*) ist gegeben durch:

$$h(t) = \sum_{n=1}^n (1 + \lfloor \frac{t - D_i}{T_i} \rfloor) \cdot C_i \quad (2.3)$$

also durch die Menge an angeforderter Rechenzeit von allen Aktivierungsinstanzen (Jobs) der Tasks, deren Aktivierungszeiten und Fristen im Intervall $[0, t]$ liegen. Die Funktion $h(t)$ wird als **Rechenzeitanforderungsfunktion** (*Demand Bound Function*) bezeichnet und repräsentiert die kumulative Anforderung der gesamten Taskmenge im entsprechenden Intervall $[0, t]$.

Abbildung 2.6: Graphische Interpretation der Rechenzeitanforderungsfunktion (DBF)

Für EDF muss dieser Test für alle Intervalle $[0, t]$ mit $t \in \mathbb{R} \leq H$ innerhalb der Hyperperiode der Tasks durchgeführt werden. Da es prinzipiell unendlich viele solche Intervalle geben kann, besitzt dieser Planbarkeitsnachweis eine exponentielle Komplexität $\mathcal{O}(k^n)$.

Die graphische Interpretation der Rechenzeitanforderungsfunktion (DBF) ist in Abbildung 2.6 anhand der im vorherigen Kapitel 2.1.1 eingeführten Beispiel-Taskmenge $\tau \in \Gamma = (T, C, D)$ mit $\tau_1 = (5, 2, 4)$ und $\tau_2 = (3, 1, 2)$ dargestellt. Die DBF hat die Form einer Treppenfunktion, deren Werte sich nur in bestimmten Unstetigkeitsstellen verändern und dazwischen konstant bleiben. Auffällig ist auch die *Winkelhalbierende* $h(t) = t$, welche die maximale Rechenzeitanforderung in Abhängigkeit von der Zeit t darstellt. Die Planbarkeitsbedingung in der Gleichung 2.4 ist dann erfüllt, wenn die DBF an keinem Punkt innerhalb der Hyperperiode H der Tasks die Winkelhalbierende übersteigt.

Zur Verminderung der Anzahl der notwendigen Planbarkeitstests und somit der Komplexität des Planbarkeitsnachweises schlagen Baruah et al. in [28] vor, die Testintervalle nur für die Punkte innerhalb der Task-Hyperperiode H zu beschränken, an denen die DBF sich tatsächlich ändert. Wie in [28] bereits gezeigt wurde, sind diese Punkte koinzident mit den Fristen der Tasks aus der gegebenen Taskmenge. Wenn jedoch die Perioden der Tasks relativ prim zueinander ausfallen, dann kann ihre Hyperperiode sehr groß werden, so dass immer noch eine sehr hohe

Anzahl an Planbarkeitstests notwendig ist. Zur Einschränkung der Testgrenze und somit der Komplexität des Planbarkeitsnachweises schlagen Baruah et al. ein Approximationsverfahren für die Rechenzeitanforderungsfunktion mit einer Funktion $C_D(t)$ vor.

Die Funktion $C_D(t)$ soll stetig sein und für alle Zeitpunkte t soll $C_D(t) \geq h(t)$ gelten — die Approximationsfunktion soll also stets größer oder gleich der DBF sein. Eine solche Funktion erhält man zum Beispiel aus der Formel der Rechenzeitanforderungsfunktion (vgl. Gleichung 2.3), wenn das Modulo aus der Formel entfernt wird:

$$C_D(t) = \sum_{i \in \Gamma} \left(1 + \frac{t - D_i}{T_i}\right) \cdot C_i \quad (2.5)$$

Durch eine entsprechende Umstellung der Formel der Approximationsfunktion:

$$\begin{aligned} C_D(t) &= \sum_{i \in \Gamma} \left(\frac{t}{T_i} + \frac{D_i + T_i}{T_i}\right) \cdot C_i = t \cdot \sum_{i \in \Gamma} \frac{C_i}{T_i} + \sum_{i \in \Gamma} \frac{C_i}{T_i} \cdot (T_i - D_i) \\ &= t \cdot \underbrace{U}_a + \underbrace{U \cdot \max(T_i - D_i)}_b \Rightarrow C_D(t) = a \cdot t + b \end{aligned} \quad (2.6)$$

sieht man leicht, dass die Approximationsfunktion die Form einer Gerade $C_D(t) = a \cdot t + b$ hat mit den Termen $a = U$ und $b = U \cdot \max(T_i - D_i)$. Der Faktor a entspricht also der Systemauslastung U . Die Konstante b wird definiert durch den maximalen Abstand zwischen der Periode und der Deadline einer Task, multipliziert mit der aktuellen Auslastung.

Da in Gleichung 2.2 die Auslastung im System $U \leq 1$ als notwendige Bedingung für die Planbarkeit einer Taskmenge definiert wurde, welche bei einem Planbarkeitstest zuerst überprüft wird, hat also die Approximationsgerade der DBF eine Steigung von $a \leq 1$ also kleiner-gleich der Steigung der Winkelhalbierenden $h(t) = t$ (vgl. Abbildung 2.7). Der Term b bestimmt die horizontale Verschiebung der Gerade im Koordinatensystem und somit auch ihren Schnittpunkt mit der Zeitachse t . Aus $b = U \cdot \max(T_i - D_i)$ folgt, dass für eine Auslastung $U \leq 1$ und Task-Fristen kleiner-gleich als ihre Perioden $D_i \leq T_i$ die Gerade eine Verschiebung im negativen Zeitbereich haben muss. Folglich muss für die genannten Bedingungen ein Schnittpunkt t' zwischen der Approximationsgerade $C_D(t)$ und der Winkelhalbierenden $h(t) = t$ existieren, ab dem die Funktionswerte der Gerade bei steigendem t immer kleiner-gleich der Winkelhalbierenden bleiben (vgl. Abbildung 2.7). Um den Wert des Schnittpunkts zu ermitteln, setzt man die Geradengleichung $C_D(t)$ gleich der Winkelhalbierenden $h(t) = t$, und mit der folgenden algebraischen Umformung nach t ergibt sich dann t' als:

$$t = t \cdot U + U \cdot \max(T_i - D_i) \Rightarrow$$

$$\begin{aligned}
 t \cdot (1 - U) &= U \cdot \max(T_i - D_i) \Rightarrow \\
 t' &= \frac{U}{1 - U} \cdot \max(T_i - D_i)
 \end{aligned} \tag{2.7}$$

Da die Gerade mit $C_D(t) \geq h(t)$ die DBF approximiert ist somit gewährleistet, dass die DBF ab t' niemals die Planbarkeitsbedingung $h(t) \leq t$ verletzen wird. Somit ist t' eine neue Testschranke für den Planbarkeitsnachweis über die Rechenzeitanforderung der Tasks. Demnach muss der Planbarkeitstest nur für die Zeitpunkte der Task-Fristen bis zur Testgrenze t' betrieben werden.

Abbildung 2.7: Approximation der DBF

Wenn die Auslastung im System hoch ist und der Abstand zwischen Fristen und Perioden der Tasks relativ groß ausfällt, kann die Testgrenze t' auf der Zeitachse weit nach hinten über die Hyperperiode der Tasks hinausfallen. In diesem Fall ist die Testgrenze $= \min(t', H)$.

Somit kann der vollständige Planbarkeitsnachweis über die Rechenzeitanforderung der Tasks wie folgt definiert werden:

Definition: Eine Menge von Tasks $\tau_i \in \Gamma$ mit $D_i \leq T_i$ ist planbar wenn,

$$\forall t \in \Lambda \quad h(t) = \sum_{n=1}^n (1 + \lfloor \frac{t - D_i}{T_i} \rfloor) \cdot C_i \quad \text{mit} \tag{2.8}$$

$$\Lambda = t \in \mathbb{R}^+ \mid \forall \tau_i \in \Gamma : t = k \cdot T_i + D_i \wedge t \leq \min(t', H) \quad (2.9)$$

d.h. für alle auftretenden Fristen der Tasks bis t' oder der ganzen Hyperperiode H nachgewiesen wird, dass die durch alle Tasks angeforderte Rechenzeit kleiner-gleich als das jeweilige Test-Intervall $[0, t]$ ist. Der Fall dass die Task-Fristen größer als die Task-Perioden sind, also $D_i \geq T_i$ bedarf ebenfalls einer speziellen Behandlung, wird jedoch im Rahmen dieser Arbeit außer Acht gelassen, da er typischerweise in realen Systemen nur selten vorkommt.

Der Algorithmus von Baruah hat eine pseudo-polynomielle Komplexität, wenn die Auslastung des Systems konstant begrenzt ist. Seine Laufzeit hängt jedoch auch vom Verhältnis der Unterschiedlichen Perioden und Fristen innerhalb der Task-Menge ab. Wenn in dieser Menge gleichzeitig Tasks mit großen und mit kleinen Perioden enthalten sind, verlängert sich die Laufzeit des Verfahrens [29].

Albers und Slomka stellen in [29] einen sehr effizienten hinreichenden Planbarkeitstest für präemptive EDF-gesteuerte Systeme vor, genannt *All Approximated*. Bei diesem Verfahren werden die Rechenzeitanforderungen der Tasks iterativ innerhalb aufsteigender Intervall-Grenzen mit einem konstanten, auslastungs-abhängigen Faktor approximiert, solange die Intervall-Grenzen durch die Approximationen nicht verletzt werden. Dieses Verfahren wird Approximation durch Superposition genannt. Wenn die approximierte DBF größer als das jeweilige Test-Intervall wird, wird ihre Approximation verfeinert und eine neue Schätzung mit den realen Kosten der Tasks etabliert. Der Test terminiert, wenn in einem Testintervall alle Tasks erfolgreich approximiert werden können. Während bei anderen Superpositions-Verfahren ein Trade-off zwischen Präzision der Approximation – also Akzeptanzrate der planbaren Task-Mengen – und Laufzeit des Algorithmus getroffen werden muss, erzeugt der *All Approximated*-Algorithmus einen dynamischen Fehler. Außerdem generiert der Algorithmus für bestimmte Task-Mengen implizit eine kürzere Testgrenze als die bisher bekannten Verfahren.

2.2 Verwandte Arbeiten

Anhand der zuvor präsentierten theoretischen Grundlagen werden nun im Folgenden ausgewählte Arbeiten vorgestellt und bewertet, die speziell eine dynamische Ressourcenvergabe an Echtzeit-Tasks oder -Anwendungen ermöglichen. Dabei stehen insbesondere solche Arbeiten im Fokus, die auch eine Anpassung der von den Tasks erzeugten Arbeitslast durch Steuerung ihrer zur Laufzeit erbrachten Qualität ermöglichen. Die Betrachtung der verwandten Arbeiten in diesem Kapitel richtet sich demzufolge nach den im Kapitel 1.3 vorgestellten Prämissen und Anforderungen

an ein ARTOS-System, das mit einer dynamischen Menge von Anwendungen und Tasks mit schwankenden Ressourcenanforderungen umgehen muss.

Können Unterversorgungen gelegentlich toleriert werden, wie in weichen Echtzeitsystemen, so lassen sich die Ressourcen effizienter nutzen, z.B. durch Scheduling anhand mittlerer (ACET) oder geschätzter Ausführungszeiten [22]. Durch Fehlschätzungen, Abweichungen oder äußere Einflüsse kann es jedoch zu Ressourcenengpässen und damit zu Verletzungen der Anforderungen kommen. Um diese selbst bei stark dynamischen Systemen unter Kontrolle zu halten, bedarf es eines kontinuierlichen Ressourcenmanagements. Dazu lassen sich zwei Ansatzmöglichkeiten identifizieren:

1. *flexible adaptive Ressourcenvergabe* und
2. *Anpassung des Ressourcenverbrauchs auf der Anwendungs- bzw. Task-Ebene.*

2.2.1 Reservierungsbasierte Verfahren

Einer der ersten reservierungs-basierten Scheduling-Mechanismen für Echtzeitsysteme wurde von Mercer et al. vorgestellt. Bei den sog. *Processor Capacity Reserves* [30] handelt es sich um im Voraus ermittelte Rechenzeit-Kapazitäten für periodische Prozesse, die zur Laufzeit von einem Betriebssystem³ durchgesetzt werden. Auf diese Weise werden die Prozesse voneinander isoliert. Eine dynamische Adaption der Reservierungen wird hier jedoch nicht unterstützt. Das Modell implementiert lediglich eine Admissions-Kontrolle anhand eines einfachen, online betriebenen Auslastungstests. Dabei werden nur Prozesse mit Deadlines gleich ihren Perioden ($D = T$) unterstützt. Der Test wird für ein zugrunde liegendes Rate Monotonic- und ein EDF-Scheduling vorgestellt. Darüber hinaus werden jedoch keine Ansätze für eine explizite Behandlung von schwankenden Ausführungszeiten bereitgestellt. Wenn ein Prozess innerhalb einer Periode mehr Rechenzeit benötigt als für ihn reserviert wurde, wird der verbleibende Anteil seiner Ausführungsinstanz im Hintergrund ausgeführt, wodurch seine Fertigstellung unvorhersehbar lang verzögert werden kann [31].

Innerhalb der sog. *Ressource Kernels* [10] erweitern Rajkumar et al. den Ansatz von von Mercer um adaptive Kapazitäts-Reservierungen. Bei diesem Modell existiert jedoch keine zentrale Entscheidungs-Direktive für die Anpassungen der Kapazitäten. Diese werden allein durch die Anwendungen selbst gesteuert und über eine Schnittstelle im Ressource Kernel angefordert, zum Beispiel wenn ihre aktuellen Kapazitäten nicht ausreichen und sie unter Fristverletzungen

³Das Modell wurde für *Real-Time Mach* implementiert.

leiden. Die Anforderungen können dabei vom Kernel gewährt oder verweigert werden. Der Kernel unterstützt ein Rate Monotonic-Scheduling für Prozesse mit Deadlines kleiner oder gleich ihren Perioden ($D \leq T$) und festen Prioritäten. Bei jeder Kapazitäts-Anpassung wird ein aufwendiger Planbarkeits-Test für alle Prozesse anhand einer Antwortzeit-Analyse betrieben.

Server-Mechanismen

Eine andere Form der reservierungs-basierten Verfahren stellen Server-Mechanismen dar. Diese werden üblicherweise für die Einplanung von aperiodischen oder sporadischen Aktivitäten innerhalb periodischer Task-Mengen mit harten Anforderungen verwendet. Ein Server wird in der Regel selbst als periodische Task realisiert, die aperiodische Last mit einer bestimmten Kapazität und Rate bedient. Dabei wird generell zwischen Ansätzen mit festen oder mit dynamischen Prioritäten unterschieden. Ein *Polling Server* [32] und ein *Deferrable Server* [33, 34] sind z.B. RMS-basierte Mechanismen, die sich in ihrer Funktionsweise stark ähneln. Die Server werden mit einer hohen Priorität und einer festen, im Voraus bestimmten Kapazität ausgeführt. Diese wird bei jeder Server-Periode neu aufgeladen. Ist die Kapazität eines Servers innerhalb einer Reservierungsperiode aufgebraucht, dann wird er, und somit die bediente Task, auf einer Hintergrund-Priorität herunterskaliert. Der Polling Server überprüft die Verfügbarkeit ausstehender Task-Jobs nur zu Beginn seiner Periode. Ungenutzte Kapazitäten (*Slack Time*) können somit verschwendet werden, wenn Task-Jobs später eintreffen, da ihre Ausführung bis zur nächsten Server-Periode verzögert wird. Ein gerade inaktiver Deferrable Server mit noch verfügbarer Kapazität kann dagegen sofort aktiv werden, wenn neue Task-Jobs eintreffen, wodurch auch eine bessere Reaktionsfähigkeit erreicht wird. Beide Server-Mechanismen werden als „Bandwidth Preserving Algorithms“ bezeichnet, da sie in der Lage sind ihre Kapazität beizubehalten, wenn keine ausstehenden Tasks existieren. Lehoczky und Ramos-Thuel entwickelten dagegen einen Mechanismus, genannt *Slack Stealer* [35], mit dem übrig-gelassene Kapazitäten (periodischer Tasks) anderen (aperiodischen) Aktivitäten verfügbar gemacht werden. Dieser Algorithmus definiert eine optimale untere Schranke für die Antwortzeiten aperiodischer Tasks innerhalb eines RMS-basierten Systems.

Die vorgestellten Server-Mechanismen basieren auf einem raten-monotonen Scheduling, das eine statische Prioritätsanordnung voraussetzt und eine volle Prozessor-Auslastung nicht erreichen kann (vgl. Kapitel 2.1). Das Modell von ARTOS benötigt jedoch eine dynamische Prioritätsverteilung, da sich die Menge der Tasks zur Laufzeit verändern kann. Soll darüber hinaus eine volle Prozessor-Auslastung von 100% ermöglicht werden, sind Mechanismen mit dynamischen Prioritäten notwendig, wie z.B. EDF. Das Scheduling-Modell von ARTOS wurde auf der Basis eines

Fixed Priority-Schedulers implementiert (vgl. Kapitel 4), wobei EDF anhand von Skalierungen der Task-Prioritäten im festen Prioritätsraum emuliert wurde.

Ein *Total Bandwidth Server* [36] (TBS) ist definiert durch ein Auslastungs-Faktor $U_s = \frac{C_s}{T_s}$, also durch eine Server-Kapazität C_s pro Server-Periode T_s , und basiert auf einem EDF-Scheduler. Der TBS-Mechanismus ist sehr simpel und besagt, dass wenn der k -te aperiodische Job zum Zeitpunkt r_k in einem TBS eintrifft, dieser dann mit einer Server-Deadline

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_s} \quad (2.10)$$

bedient wird. Dabei sind C_k die benötigte Ausführungszeit (WCET) des Jobs, d_{k-1} die letzte Server-Deadline und U_s der Auslastungs-Faktor (oder Bandbreite) des Servers. Der Job wird dann entsprechend seiner Deadline in die Warteschlange eines EDF-Schedulers eingereiht. Die dynamische Deadline-Zuweisung stellt sicher, dass die Beiträge der durch einen TBS bedienten aperiodischen Jobs zum Auslastungs-Faktor des Servers niemals größer als U_s werden, wenn die Jobs nicht länger als ihre deklarierten WCETs C_k ausführen [31].

Um temporale Isolation zwischen den bedienten Tasks auch bei Überläufen (*Overruns*) zu ermöglichen, wurden von Abeni et al. auf der Basis der TBS die sog. *Constant Bandwidth Server* [31] (CBS) entwickelt. Diese dienen als Grundlage des hierarchischen Scheduling-Modells von ARTOS und werden darum im Kapitel 3.2.2 intensiv behandelt. Dabei wurden die CBS, die zunächst task-basiert arbeiten, für Kapselung und Isolation ganzer Anwendungen mit mehreren periodischen Tasks erweitert und angepasst. Die CBS implementieren dabei ein internes EDF-Scheduling für die bedienten Tasks. Für CBS existieren ebenfalls bereits dynamische Anpassungsmechanismen ihrer Kapazitäten. Durch *Resource Reclaiming* [37, 38] können zum Beispiel konservative Annahmen über den Ressourcenbedarf einer bedienten Task zur Laufzeit dynamisch reduziert werden. Ungenutzte Kapazitäten können auch durch einen *Capacity Sharing*-Mechanismus (CASH) [39] anderen Tasks verfügbar gemacht werden und sorgen damit dafür, dass diese weniger wahrscheinlich in Engpässe geraten. In ARTOS wurden darüber hinaus Mechanismen entwickelt, mit denen die Kapazitäten der CBS dynamisch aus den zur Laufzeit ermittelten, schwankenden Rechenzeit-Anforderungen der gekapselten Tasks abgeleitet und ihnen zugewiesen werden können.

Feedback-Scheduling

Feedback-Scheduling [40, 41, 42] arbeitet ebenfalls reservierungsbasiert. Eine Regelung passt die Ressourcen dynamisch an, so dass die genauen Anforderungen nicht vorab bekannt sein

müssen. Ähnlich wie in dieser Arbeit ist die Idee dabei, das aggregierte Verhalten eines Systems makroskopisch zu erfassen statt alle Vorgänge darin im Detail zu modellieren und zu planen [22]. Der Unterschied zum hier präsentierten Ansatz ist, dass das hierarchische Scheduling-Modell von ARTOS dabei auch automatisch für temporale Korrektheit bei der Planung und Ausführung der einzelnen Vorgänge sorgt.

Beim Feedback-Scheduling adaptiert ein Regelkreis Scheduling-Parameter zur Laufzeit anhand des beobachteten Task-Verhaltens. Manche Strategien verwenden dazu globale Metriken als Steuergrößen, z.B. die beobachtete Deadline-Verletzungsrate oder die totale Ressourcenauslastung im System [43, 44, 45]. Da diese Ansätze recht grobgranular sind und nur eine globale Sicht auf die Qualität eines Systems haben, fließt das individuelle Verhalten einer Task nur indirekt in die Entscheidungen ein [46]. Solche Scheduler können verfügbare Ressourcen nicht gezielt an Engpass-Stellen verteilen, um stationäre Unterversorgungen oder Ressourcenüberschüsse entsprechend zu behandeln.

Abeni et al. stellen einen (task-lokalen) Feedback-Scheduler vor [47, 48], bei dem das Verhalten jeder Task isoliert betrachtet und periodisch für dynamische Reservierungen verwendet wird. Dabei werden Tasks in einzelnen CBS gekapselt und anhand ihrer Server-Deadlines mit EDF aktiviert. Sie können jedoch auch eigene Fristen haben, die eingehalten werden müssen (das Modell impliziert dabei nur Tasks mit $D = T$). Ein Regelkreis reguliert dynamisch die allozierten Server-Kapazitäten (und somit Bandbreiten) so, dass ein minimaler Scheduling-Fehler entsteht. Der Scheduling Fehler ist definiert als die Distanz zwischen der aktuellen Server-Deadline und der eigenen internen Frist einer Task. Er repräsentiert die Differenz zwischen der durch die Server-Deadline garantierte Frist und die eigene zu erfüllende Frist der Task. Eine Bandbreitenanpassung findet bei jeder Task-Periode statt. Dabei wird die Bandbreite erhöht, wenn der Scheduling-Fehler groß ist und proportional zu einer Gewichtung der Task im System verkleinert, wenn ihre Ressourcenanforderung überprovisioniert wird. Wird das System durch die dynamischen Bandbreitenanpassungen überbelastet, werden die Kapazitäten proportional nach Task-Prioritäten degradiert. Dies liefert zwar keine Garantie dafür, dass sich das System von Fristverletzungen komplett erholen kann, höher priorisierte Anwendungen leiden jedoch weniger unter dem globalen Engpass.

Abeni et al. zeigen, dass mit der vorgestellten Regelung der Scheduling-Fehler gegen Null konvergiert und minimal gehalten werden kann, wenn das System nicht überlastet ist. Somit kann prinzipiell eine Wahrscheinlichkeit für das Erfüllen der Task-Fristen abgeleitet werden. Konzepte für den Umgang mit permanenten Ressourcen-Engpässen durch Qualitätsanpassungen werden nicht vorgestellt. Ebenso wäre es interessant, wie hoch der minimale Scheduling-Fehler innerhalb der präsentierten Experimente ausfallen würde, wenn die Tasks kleinere Fristen als ihre Perioden

haben. Hierbei ermöglicht das Modell von ARTOS durch eine dynamische Planbarkeitsanalyse eine Kapazitätsbemessung für die Tasks einzelner in CBS gekapselten Anwendungen, die das Einhalten task-interner Fristen auch bei $D \leq T$ sicherstellt, solange die Task-Kosten nicht zu stark schwanken (siehe Kapitel 3.2.2). Wenn Letzteres der Fall ist, dann werden neue Kapazitäten ausgerechnet und potentielle permanente Engpässe durch gezielte Degradierung der Qualität von Anwendungen in unterschiedlichen Betriebsmodi umgangen.

Diskussion

Mechanismen, die allein auf der Ebene einer adaptiven Ressourcenvergabe arbeiten, sind gut geeignet, transiente Ressourcenengpässe einzelner Tasks durch Reallokation ungenutzter Ressourcen zu überwinden. Sie stellen jedoch per se keine Mechanismen zum Umgang mit persistenten Engpässen aufgrund fehlender Gesamtressourcen zur Verfügung. Wenn durch Veränderungen des Ressourcenverbrauchs der Tasks oder durch eine Erweiterung des Applikations-/Task-Umfangs innerhalb eines laufenden Systems die verfügbaren Ressourcen überalloziert werden, dann leiden unter Umständen alle Anwendungen unter permanenten Anforderungsverletzungen [49, 12]. Durch den Einsatz von Constant-Bandwidth-Servern können die Auswirkungen von Kosten-Überläufen zwar auf die verursachenden Aktivitäten eingeschränkt werden, generelle Degradierungsmöglichkeiten der Komplexität und somit des Ressourcenverbrauchs der Anwendungen werden in den bisher vorgestellten Arbeiten nicht angeboten.

2.2.2 Methoden zur Überlastbehandlung

Allgemein existieren drei grundlegende Ansätze zur Behandlung persistenter Überlast durch Adaption auf der Anwendungsebene [22]: Auslassen von Jobs periodischer Tasks (*Job Skipping* [50]), Vergrößerung von Taskperioden (*Elastic Scheduling* [51]) und Herabstufung der Qualität und somit des Rechenaufwands der Tasks (*Quality Adaptation*). Hier gibt es typischerweise einen Zielkonflikt zwischen niedrigen Verletzungsraten, guter Qualität und guter Ressourcenauslastung bei schlechter Vorhersagbarkeit des Ressourcenverbrauchs.

Alle drei Ansätze können durch die Einführung unterschiedlicher Qualitätsstufen von Tasks (bzw. Applikationen) nachgebildet und kombiniert werden. Dabei implementieren die Anwendungsentwickler Anpassungsmöglichkeiten in Form unterschiedlicher *Betriebsmodi* der Anwendung, in denen Jobs ausgelassen, Task-Perioden modifiziert und die Berechnungskomplexität der Tasks reduziert werden kann. Die Modi abstrahieren also die genaue technische Umsetzung der Degradierung und erlauben die Definition von einem modus-spezifischen *Nutzen*. Aufgabe eines

entsprechenden Schedulers ist es, optimale Modikonfigurationen zu generieren, so dass mit den vorhandenen Ressourcen der bestmögliche Nutzen erzielt wird. Diese Herangehensweise resultiert in ganz neuen Anforderungen an System und Scheduler, auf deren Folgen der wissenschaftliche Fokus zunächst gelenkt werden soll.

Bereits Tokuda und Kitayama diskutierten unterschiedliche Betriebsmodi für Anwendungen und eine qualitätsorientierte Regelung des Ressourcenverbrauchs [52]. QoS-Stufen sollen als Basis für Kapazitätsreservierungen für Prozessoren [30] dienen und werden durch Periodenänderungen von Tasks bzw. durch Modifikation ihrer Rechengenauigkeit charakterisiert. Die Autoren haben jedoch weder präzise Schedulingmodelle ausgearbeitet noch ein funktionierendes System entwickelt [53].

Imprecise Computations

Mit **Imprecise Computations** (IC) entwickelten Liu et al. [54, 55] den ersten theoretisch fundierten Ansatz Tasks in mehreren Betriebsmodi aufzuteilen. Hierfür stellen die Autoren mehrere Implementierungstechniken vor, wie z.B. *Sieve Functions* oder *Multiple Versions*⁴. Ein rechtzeitiges Resultat von geringerer Qualität wird einem verspäteten Resultat mit optimaler Qualität vorgezogen. Eine Task besteht aus mehreren Jobs, die jeweils in einen Pflichtteil (*Mandatory Part*) und einen Wahlteil (*Optional Part*) zerlegt werden, jedoch dieselbe Deadline haben. Die vom Pflichtteil erbrachte Mindestqualität wird durch die Ausführung des Wahlteils verbessert. Das Scheduling von IC ist so ausgelegt, dass die Pflichtteile aller Tasks in jedem Fall ihre Deadlines einhalten können; eine Verletzung dieser Anforderung wird durch Admissionskontrolle bei Kenntnis der Task-WCET verhindert. Die verbleibende CPU-Zeit wird danach unter den optionalen Teilen optimal aufgeteilt, was in der Regel ein NP-hartes Problem ist. Daher werden eine Reihe heuristischer Verfahren vorgeschlagen [54]. Die repräsentativsten Ansätze versuchen den durchschnittlichen Fehler zu minimieren, welcher durch unpräzise Ergebnisse erzeugt wird. Dabei werden den Tasks dynamisch Reservierungen vergeben, innerhalb welcher jedoch die Ausführung der Wahlteile jederzeit abgebrochen werden kann. Andere Ansätze versuchen wiederum optionale Teile mit einer bestimmten Wahrscheinlichkeit als Pflichtteile einzubringen, zum Beispiel, wenn der kumulative Fehler ansteigt. Diese Verfahren eignen sich ideal für transiente Überlastsituationen oder wenn die tatsächlichen Ausführungszeiten der Tasks stark variieren können. Sie sind jedoch nicht optimal, d.h. unter Umständen nicht in der Lage einen präzisen, gültigen Plan für Task-Mengen zu erzeugen, die mit klassischen RMS bzw. EDF Strategien planbar sind [55].

⁴Auf die algorithmischen Realisierungsmöglichkeiten unterschiedlicher Betriebsmodi wird später im Kapitel 2.4 noch einmal genauer eingegangen.

Während die meisten IC-Ansätze deterministische Ausführungszeiten voraussetzen, untersuchten Chong et al. probabilistische Ausführungszeiten [56]. Sie gehen jedoch von unterschiedlichen stochastischen Verteilungen der Ausführungszeiten aus (z.B. exponentiell), was nicht immer eine realistische Annahme über das Task-Verhalten sein muss. Task-Anforderungen können nicht-deterministisch variieren, so dass keine vordefinierte Verteilung diese vollständig approximiert. Hamann et al. erweitern das IC-Task-Modell um mehrere optionale Teile pro Task und gehen von statistisch verteilten Ausführungszeiten aus, die von Worst-Case-Zeiten begrenzt sind [57]. Dasselbe Task-Modell wird auch von Härtig et al. in [58] für *container-gesteuerte Qualitätsanpassungen*⁵ verwendet. Vordefinierte *Qualitätsprofile* von Komponenten mit geschätzten Ressourcen- und Qualitätsangaben werden zentral mit einem Container verhandelt. Entscheidungsvorgänge für Qualitätsanpassungen⁶ aufgrund realer Verbräuche zur Laufzeit, werden jedoch nicht näher beschrieben. Andere vorgestellte Adaptionsmechanismen benötigen Vorabmessungen durch die Entwickler.

Die Ressourcenverteilung bei IC findet nach einem Greedy-Verfahren statt. Zuerst werden alle Pflichtteile eingebracht und danach werden die übrigen Ressourcen unter den Wahlteilen der Tasks aufgeteilt. Dieser Ansatz ist gegenüber dem hier vorgestellten hierarchischen Entscheidungsmodell komplementär. Das Ziel der Ressourcenkontrolle bei ARTOS ist, jede Anwendung in einen optimalen Betriebsmodus zu schalten. Die Betriebsmodi zwingen einen Entwickler jedoch nicht in die Zerlegung in Pflicht- und Wahlteile. Stattdessen kann jeder Modus zielgerichtet entwickelt werden. Trotzdem können IC-basierte Anwendungen leicht in ARTOS überführt werden. Im Gegensatz zu IC arbeitet ARTOS ohne Vorkenntnisse über die Bedarfe. Die Modi werden anhand dynamisch approximierter Ressourcenbandbreiten von Anwendungen zur Laufzeit eingestellt. Die Planbarkeit der applikations-lokalen Task-Mengen wird innerhalb dieser Bandbreiten sichergestellt. Sie wird auch nur dann aktualisiert, wenn Anwendungs-Tasks aus entsprechenden Approximationsgrenzen ausbrechen und nicht bei jeder Task- oder Scheduler-Periode. Die Online-Version von IC unterstützt eine dynamische Taskmenge und variierende Ressourcenbedarfe, entscheidet aber bei jeder Reservierungsperiode neu über die Ausführung von Wahlteilen. Im Gegensatz dazu wird bei ARTOS davon ausgegangen, dass das Umschalten eines Modus zusätzliche Anpassungskosten verursacht und ständige Rekonfigurationen daher zu vermeiden sind.

⁵Container-Managed Quality Assurance

⁶z.B. durch Transitionen zwischen Qualitätsstufen (*Subprofile* [58]) oder Auslassen optionaler Teile

Hybride regelungstheoretische Ansätze mit Qualitätsanpassungen

Stankovic, Lu et al. präsentieren einen hybriden Ansatz zwischen feedback- und anwendungs-basierter Adaption [44, 45]. Mechanismen aus der Regelungstheorie werden für dynamische Taskplanung und Admission in einem Echtzeitsystem angewandt. Diese bilden die theoretische Grundlage des sog. *Feedback Control Real-Time Scheduling* (FCS). Der **FCS-Scheduler** misst die globale Deadline-Verletzungsrate (*Miss Ratio*) und den totalen CPU-Verbrauch (*Utilization*) und schleift diese als Steuervariablen an einen (PID-)Regler zurück. Das Ergebnis der Regelung ist eine Korrektur der Gesamtauslastung des Systems, wobei Sollwerte der Steuervariablen respektiert werden. Ein QoS-Aktuator soll damit den Ressourcenverbrauch der internen Tasks anpassen. Als Anwendungsmodell wird hier der IC-Ansatz [55] vorgeschlagen. Ein nutzenbasierter Optimierungsalgorithmus im Aktuator berechnet mit dem Ergebnis der Regelung die neue Konfiguration der Task-Qualitäten und betreibt Admission. Hierfür verweisen die Autoren auf bereits existierende Lösungsmethoden des Optimierungsproblems beim IC-Modell.

Der wissenschaftliche Beitrag von FCS liegt in der analytischen Modellierung und im Tuning der Regelung. Besonders bei der Linearisierung der Steuervariablen für unterschiedliche Sättigungspunkte wird auf Heuristiken zurückgegriffen. Beispielsweise wird durch einen zunächst experimentell bestimmten Deadline-Factor eine lineare Beziehung zwischen CPU-Auslastung und Deadline-Miss-Rate angenommen. Über den Sättigungspunkt der Regelung sind jedoch keine weiteren Aussagen möglich. Verschiedene Parameter der Regelung werden experimentell bestimmt, um sicher zu stellen, dass sich das System bei Überlast wieder stabilisieren kann. Da keine speziellen Algorithmen zur Approximation der Taskverbräuche und zur optimalen Ressourcenverteilung vorgestellt wurden, leidet der FCS-Ansatz unter den selben Nachteilen wie der IC-Ansatz.

Cucinotta et al. präsentieren in [53] eine Erweiterung des an der SSSUP⁷ entwickelten und zuvor präsentierten reservierungs-basierten Schedulers [48] von Abeni et al. für Multi-Mode-Anwendungen. Ihr hybrider Ansatz besitzt zwei konkurrierende Feedback-Schleifen zur Ressourcenkontrolle. Ein task-lokaler Regler (*Task Controller*) justiert die jeweilige Ressourcenbandbreite. Da er jedoch nur eine task-lokale Sicht hat, kann es zu Engpässen kommen. Ein globaler QoS-Regler überprüft periodisch die Systemauslastung und konfiguriert gegebenenfalls die Tasks in niedrigere Betriebsmodi. Das in dieser Arbeit präsentierte Anwendungs- und Scheduling-Modell ist dem hierarchischen Modell von ARTOS konzeptionell ähnlich. In ARTOS werden jedoch die Regler durch eine intelligente Vorverarbeitung gemessener Werte, durch eine Planbarkeitsanalyse

⁷Scuola Superiore Sant’Anna di Studi Universitari e di Perfezionamento

zur Kapazitätsbemessung von Anwendungen und eine anschließende Optimierung der Modifikation ersetzt. Der Augenmerk liegt hierbei auf der Minimierung und zeitlichen Verteilung des dabei entstehenden Overheads und auf der Vermeidung von Rekonfigurationen, da hierbei zusätzliche Kosten entstehen. Im Gegensatz zu Cucinotta et al. berücksichtigt ARTOS auch den Jitter und periodische Bursts von Tasks durch entsprechende Approximations-Mechanismen. In [53] werden iterative heuristische Verfahren mit einer Mehrwegsuche zur Vermeidung von lokalen Optima eingesetzt, was häufige Rekonfigurationen bis zum stabilen Zustand zur Folge hat. Die Optimierung der Anwendungsmodi in ARTOS zeigt eine ähnliche Laufzeit, erreicht jedoch zielgerichtet eine global optimale Lösung (was später in den Kapiteln 3.2.3 und 5 gezeigt wird).

Q-RAM basierte Arbeiten

Bei **Q-RAM** [59] können Benutzer spezielle Anforderungen (*Quality Points*) innerhalb mehrerer Qualitätsdimensionen spezifizieren und subjektiv den Nutzen (*Utility*) beim Erfüllen bestimmter Qualitätsanforderungen für jede Anwendung angeben. Die optimale Verteilung der verfügbaren Ressourcen wird dann auf ein multidimensionales Optimierungsproblem reduziert, welches von Q-RAM mittels Linearer Programmierung zur Laufzeit gelöst wird. Als Basis der Optimierung dienen so genannte Task-Profile, welche unter anderem Abbildungsfunktionen zwischen Ressourcen und der damit erreichten Qualität innerhalb einer Anwendungsdimension beinhalten. Die Ressourcenzuteilung versucht den maximalen Nutzen für alle Anwendungen zu erzielen und dabei die Mindestanforderungen der Anwendungen nicht zu verletzen. Wenn das Ergebnis der Optimierung die Mindestanforderungen einer Anwendung nicht mehr erfüllen kann, wird diese abgeschaltet. In diesem Sinn eignet sich Q-RAM gut für eine feingranulare Admissionskontrolle für Anwendungen in einem weichen Echtzeitsystem.

Ecker et al. gehen in [60] auf Veränderungen der verfügbaren Gesamtmenge an Ressourcen in einem Q-RAM-basierten System ein. Sie stellen eine Methode für daraus abgeleitete Reallokationen bestehender Ressourcenreservierungen der Anwendungen innerhalb der einzelnen Qualitätsdimensionen vor.

Ghosh et al. verwenden in [61] Q-RAM für ein den Nutzen maximierendes Task-Scheduling innerhalb eines dynamischen Echtzeitsystems. Eine dynamisch variierende Taskmenge fordert regelmäßige Reallokationen bereits zugewiesener Ressourcen. Q-RAM wird für die Berechnung einer optimalen Ressourcenverteilung an die Tasks eingesetzt. In einem iterativen Prozess wird versucht, ein vorgegebenes Nutzenniveau zu erreichen. Nach einem Optimierungslauf wird überprüft, ob für die ermittelte Zuteilung genug Ressourcen vorhanden sind. Wenn dies nicht möglich

ist, wird das Nutzenniveau abgesenkt und der Vorgang mittels einer Feedback-Schleife wiederholt.

Harada et al. führen in einem Q-RAM ähnlichen Modell *Fair QoS* Fairness-Aspekte zwischen Tasks ein [62]. Damit wird verhindert, dass Ressourcenverteilungen ausgewählt werden, die rechnerisch optimal sind, jedoch einzelne Tasks benachteiligen. Es wird für jede Task, unter Betrachtung der Ressourcenanforderungen der besten und der schlechtesten Stufe, eine mittlere (faire) Qualitätsstufe ausgewählt. Als Kriterium gelten ähnlich wie bei Q-RAM Ressourcen-Qualitätsabbildungen. Ein Online-Algorithmus mit polynomialer Komplexität berechnet damit eine faire Verteilung der Ressourcen, wobei die Minimierung der Distanz zu den mittleren Qualitätsstufen der Tasks angestrebt wird.

Trotz Ähnlichkeiten von Q-RAM und seinen Derivaten existieren grundlegende Unterschiede zu ARTOS. Q-RAM benötigt vordefinierte Task-Profile durch vorab durchgeführte Messungen, während ARTOS ohne Vorwissen auskommt. Insbesondere wird bei ARTOS mit jitternden Ausführungszeiten umgegangen, während der Scheduler des in [61] vorgestellten Radar-Systems dagegen nahezu Jitter-Freiheit von Tasks voraussetzt. Somit kann bei ARTOS eine präzise Aussage über die Planbarkeit von Tasks auch bei starken Schwankungen ihrer Ressourcenverbräuchen getroffen werden und die Erfüllung der Echtzeitanforderungen ihrer Anwendungen garantiert werden. Ferner zielen die Q-RAM-basierten Ansätze darauf ab, eine gültige Ressourcenkonfiguration für eine vorgegebene QoS-Konfiguration zu finden. Bei ARTOS wird dagegen die optimale Konfiguration bei einer vorgegebenen und sich eventuell sogar verändernden Ressourcenmenge unter Beachtung von Benutzerpräferenzen ermittelt.

Weitere ausgewählte Arbeiten

Brandt et al. stellen eine Middleware-basierte Lösung für Anwendungen mit mehreren Qualitätsstufen vor [63]. Diese besteht im Wesentlichen aus zwei Aktuatoren, die gegeneinander arbeitend ein Optimum anstreben: Ein *QoS-Manager* reagiert auf Deadline-Verletzungen und entscheidet über die Degradierung von Anwendungen in eine niedrigere Stufe. Ein *Sampler* überwacht den totalen Ressourcenverbrauch und schaltet Applikationen in höhere Betriebsmodi falls Ressourcen übrig sind. Ferner besitzen Applikationen vordefinierte *Echtzeit-Spezifikationen* mit relativen Angaben über den CPU-Verbrauch und Qualität ihrer Modi. Diese Angaben werden dann zur Laufzeit durch Messungen verfeinert. Gemessene Ausführungszeiten der Tasks werden geglättet, um Jitter-Effekte abzuschwächen und dienen als Basis der Ressourcenoptimierung. Hierfür präsentieren die Autoren eine Erweiterung von Jensens *Benefit-based Scheduling* [64].

Dieser verwendet task-spezifische *Time-Value Functions* zur Berechnung einer Ausführungsreihenfolge, die stets eine maximale Qualität im System produziert. Dieser Ansatz leidet unter Konfigurationsinstabilität bei Anwendungen mit variierenden Anforderungen. Der verwendete Optimierungsalgorithmus generiert schon bei kleinen Änderungen seiner Vorgaben unterschiedliche Konfigurationen, was von den Autoren selbst bemängelt wird. Zudem ist die Optimierung bei ARTOS annähernd 100-Mal schneller als der von Brandt präsentierte Ansatz. Um häufige Rekonfigurationen einzudämmen, definieren die Autoren Schwellwerte für die Input-Variablen des QoS-Managers und des Samplers. Im Gegensatz dazu wird bei ARTOS das Jitter- und Burstverhalten von Tasks ermittelt und berücksichtigt.

Oliveira et al. schlagen in [65] adaptive Multi-Mode-Anwendungen vor, deren Tasks in einzelnen Constant-Bandwidth-Servern gekapselt sind. Somit birgt dieser Ansatz gegenüber ARTOS einen höheren Scheduling-Aufwand. Für jede Task ist bekannt, wieviele Ressourcen für einen erfolgreichen Lauf in jeder Anwendungskonfiguration notwendig sind. Das Modell geht jedoch von einer einzigen multi-modalen Anwendung innerhalb eines Systems aus. Kann das System nicht alle Anforderungen optimal befriedigen, wird die Auswahl der Anwendungskonfiguration für die beste Qualität des Gesamtsystems auf ein Optimierungsproblem zurückgeführt. Damit wird in erster Linie eine dynamische Admissionskontrolle implementiert. Hierfür stellen die Autoren ein approximatives Lösungsverfahren mittels Integer-Programmierung vor. Anschließend werden die Parameter der CBS an die neuen Verbräuche der Tasks angepasst. Das Konzept ähnelt dem in dieser Arbeit präsentierten Ansatz, doch benötigt ARTOS kein Vorwissen über die Ressourcenbedarfe und passt sich an deren Variationen an. Oliveira et al. argumentieren, dass die Optimierung durch ein klassisches Rucksackproblem mit dynamischer Programmierung wegen ihrer pseudo-polynomiellen Komplexität für ein solches System nicht anwendbar sei. Bei den in [65] genannten Problemstellungen mit mehreren Hunderten von Tasks erscheint dies plausibel. Bei ARTOS gehen wir jedoch nur von einer zweistelligen Anzahl aktiver Applikationen aus, was nach den Ergebnissen der Evaluation in Kapitel 5 akzeptable Laufzeiten erlaubt.

Bei **FRESCOR** werden Ressourcen anhand deklarativ und abstrakt beschriebener Anforderungen (*Contracts*) vergeben, die zur Laufzeit in konkrete Ressourcenanforderungen übersetzt werden. Aus der Literatur lassen sich zwei offenbar verschiedene Ansätze unterscheiden: Ein erster Ansatz [66] basiert auf [67] und ähnelt sehr stark den Imprecise Computations, die bereits ausführlich diskutiert wurden. Ein zweiter, verwandterer Ansatz basiert auf einem QoS-Management Modul für Multi-Mode-Anwendungen [68]. Abstrakte Application Level Contracts (ALCs), werden auf konkrete Resource Level Contracts (RLCs) abgebildet. Dabei werden statistische Informationen über das Laufzeitverhalten der Anwendungen ausgenutzt. Die Abbildung ist ein Optimierungsproblem, welches während einer Verhandlungsphase gelöst werden muss.

Der RLC beinhaltet dabei auch den ausgewählten Betriebsmodus. Das QoS-Management setzt auf ein hierarchisches Scheduling auf. Jede Ressource besitzt einen Resource Manager und einen Resource Scheduler, die durch eine Feedback-Schleife gekoppelt sind. Somit können die den Anwendungen allozierten Bandbreiten gemäß den in den RLCs spezifizierten Kapazitäten feinjustiert werden [69]. Applikations-lokale Server sorgen wiederum für Kostendurchsetzung. Wenn Qualitätsanforderungen nicht eingehalten werden können (z.B. Ressourcenengpässe) oder sich ändern (z.B. Priorisierung durch Benutzer), so kann ein QoS Broker oder eine Anwendung selbst eine Neuverhandlung der ALCs initiieren. Dabei können beliebige Anwendungen in niedrigere Modi geschaltet werden.

Bei der Übersetzung von ALCs zu RLCs werden vorab gesammelte Informationen über das Laufzeitverhalten herangezogen. Ausführungszeiten der unterschiedlichen Betriebsmodi müssen aus früheren Abläufen bekannt sein. Andernfalls werden Verbräuche geschätzt und gegebenenfalls online justiert. Rekonfigurationen können bei FRESCOR durch Overprovisioning von Ressourcen reduziert werden, wobei jedoch die Überallokationen zu grobgranular und fest sind. Bei ARTOS greifen hier adaptive task-basierte Approximationen, die Schwankungen entsprechend miterfassen. Diese werden periodisch überprüft und notfalls justiert, wodurch die Ressourcenüberprovisionierungen bei ARTOS adaptiv und abhängig vom Schwankungsverhalten der Anwendungen sind. Aus diesen Approximationen ergeben sich dann maßgeschneiderte Ressourcenkapazitäten für die einzelnen Anwendungen, mit denen ihre Echtzeitanforderungen ausgehend von ihrem aktuellen Schwankungsverhalten erfüllt werden. In ARTOS können zusätzlich periodische Bursts und Slack Time von Anwendungen geeignet gegeneinander ausgeglichen werden, so dass Rekonfigurationen gering bleiben und trotzdem nur wenig Ressourcen verschwendet werden.

Es existieren eine Fülle weiterer Arbeiten, die sich mit um Ressourcen konkurrierende Anwendungen bei vorgegebenen QoS-Anforderungen beschäftigen. Bei diesen Arbeiten sind jedoch die Spezifikationen der Ressourcenanforderungen von Tasks fix und meist fehlen auch Arbitrierungsmethoden bei der Verteilung von Ressourcen. Abdelzaher et al. schlagen zum Beispiel vor, QoS-Stufen von Tasks mit einer Reihe von Überlastbedingungen a priori zu beschreiben, um eine kontrollierte Degradierung des Systems zu erlauben [70]. Diese Methode berücksichtigt jedoch keine dynamische Auswahl der Modi. Bei Lee et al. wird zur Laufzeit einer von mehreren offline erstellten Ablaufplänen für die Schaltung von Tasks in unterschiedlichen Betriebsmodi unter Beachtung von Energieeinschränkungen ausgewählt [71]. Hier wird von probabilistischen Ausführungszeiten der Tasks pro Betriebsmodus ausgegangen, deren Verteilung jedoch offline durch Experimente ermittelt wird. Abeni und Buttazzo führen das Elastic Scheduling Model ein, das Perioden von Tasks in Überlastsituationen dynamisch wie eine Feder dehnt oder kürzt [72, 51]. Dies erfordert die Einhaltung eines speziellen Task-Modells und ist daher in der realen Umset-

zung innerhalb Anwendungen relativ komplex. Brandt et al. stellen eine Verhandlungsmethode für Ressourcenallokationen vor, welche auf Angaben von maximalen Prozessorkosten und maximalem dadurch erlangtem Benefit pro Qualitätsstufe basiert [73]. Sie entwickelten darauf basierend die bereits diskutierte Middleware für QoS-Management [63]. In [10, 74] präsentieren Rajkumar und Oikawa einen Resource Kernel mit einer uniformen Benutzer-API für Ressourcenreservierungen. Anwendungen können damit eine beliebige Menge von Ressourcen für eine bestimmte Dauer reservieren und werden benachrichtigt, falls ihre Reservierung nicht (mehr) eingehalten werden kann. Hier werden jedoch keine Ansätze für das Management von Überlastsituationen und die kontrollierte Degradierung von Applikationen präsentiert, diese sind Teil und Aufgabe der Anwendungen.

Nach dieser Abhandlung der verwandten Arbeiten werden nun im folgenden Kapitel einige Grundlagen der Video-Codierung und -Übertragung vorgestellt.

2.3 Grundlagen der Video-Codierung und -Übertragung

Multimediale Anwendungen, speziell solche, bei denen bereits gespeicherte oder zur Laufzeit über ein Netzwerk empfangene Audio-/Video-Inhalte decodiert und dargestellt werden, eignen sich gut als Anwendungsfälle für das in dieser Arbeit entwickelte hierarchische Scheduling-Modell. Einerseits verfügen solche Anwendungen über periodischen Aktivitäten mit weichen Echtzeitanforderungen, da beispielsweise bei einem Video eine bestimmte Bildwiederholfrequenz auf einem Ausgabegerät erreicht werden muss. Auf der anderen Seite enthalten verwendete Multimedia-Formate und -Codecs bereits standardisierte Maße für die Qualität der medialen Inhalte, sowie Mechanismen und Faktoren zu ihrer Festlegung und Anpassung. Solche Mechanismen werden in Kapitel 2.4 unter anderem am Beispiel von Video-Signalen untersucht und bewertet. Dabei stehen Möglichkeiten zur Realisierung von festen und zur Laufzeit dynamisch umschaltbaren Qualitätsstufen im Fokus. Aus diesem Grund werden in diesem Kapitel die grundlegenden Mechanismen bei der Codierung und Übertragung von Videos grob vorgestellt. Im Kapitel 5 wird wiederum im Rahmen der Evaluation ein Video-On-Demand Szenario präsentiert, welches mit unterschiedlichen Qualitätsmodi auf dem hierarchischen Scheduling Modell aufsetzt.

2.3.1 Video-Codierung

Bei der Video-Codierung wird ein Video mittels eines Encoders komprimiert, um seine Datenrate zu reduzieren und es so einfacher speichern und übertragen zu können. Um das Video wieder

abspielen zu können, muss es zunächst von einem Decoder verarbeitet werden. Bei der Codierung wird generell zwischen verlustlosen und verlustbehafteten Verfahren unterschieden. Bei verlustlosen Verfahren wird lediglich Redundanz aus dem Videosignal entfernt. Bei verlustbehafteten Verfahren werden dagegen Informationen aus dem Video gelöscht, sodass das ursprüngliche Signal nicht vollständig wieder rekonstruierbar ist. Damit wird jedoch eine stärkere Kompression erreicht. Bei einem Video-Streaming Szenario zum Beispiel ist die übertragene Datenmenge sehr groß. Aus diesem Grund wird bei der Übertragung eines Videos über das Internet in der Regel verlustbehaftete Kompression verwendet. Bei den verlustbehafteten Verfahren kann bestimmt werden, wie sehr die Informationsverluste vom Betrachter wahrgenommen werden können. Moderne Codecs nutzen Eigenschaften der menschlichen visuellen Informationsverarbeitung aus, um die Informationsverluste möglichst gut zu maskieren.

Der Audiostrom eines Videos wird meist mit einem separaten Audiocodec komprimiert. Im Rahmen einer Vorarbeit [75] wurde eine echtzeitfähige Implementierung eines AAC-Encoders [76] für das DROPS [77] Betriebssystem entwickelt. DROPS basiert auf den L4-Mikrokern [78], welcher das Anwendungs- und Scheduling-Modell der Imprecise Computations unterstützt. Anpassungen des Audiostroms für die Realisierung einer optimalen (Wahlteil) und einer Mindestqualität (Pflichtteil) wurden durch eine gezielte Annulierung von Frequenzbändern bei der Quantisierung des Audiosignals ermöglicht. Da die Einsparungen durch die Videokomprimierung deutlich effizienter ausfallen, liegt der Fokus in dieser Arbeit auf der Qualitätsanpassung von Video-Signalen.

Die bekanntesten Videocodecs wurden von der der ISO angehörigen *Moving Picture Experts Group* (MPEG) und der der ITU-T angehörigen *Video Coding Experts Group* (VCEG) entwickelt. Zu diesen Codecs gehören H.261 (1990), MPEG-1 Video (1993), MPEG-2 Video (1994), H.263 (1995), MPEG-4 Part 10 / H.264 (2003) und MPEG-H Part 2 / H.265 (2013). Zur Entwicklung jungerer Codecs sind beide Gruppen eine Partnerschaft eingegangen, welche als *Joint Video Team* (JVT) bekannt ist. In diesem Abschnitt sollen die grundlegenden Komprimierungsstrategien am Beispiel von H.264 erläutert werden.

Codierung einzelner Frames

Bei H.264 [79, 80] werden einzelne Frames in eine Menge von *Makroblöcken* der Größe 16×16 Pixeln unterteilt, die als Basis für die Komprimierung verwendet werden. Die einzelnen Makroblöcke werden dann einer Farbunterabtastung unterzogen, bei der sie vom RGB-Farbraum in einem YCbCr-Farbraum umgerechnet werden. Die Farbinformation wird bei diesem Farbmodell in eine Grundhelligkeit Y und zwei Farbkomponenten Cb und Cr zerlegt. Beim gewöhnlichen

Abtasten würde nun der Makroblock Pixel für Pixel abgetastet und jedem Pixel ein Wert aus diesem Farbraum zugewiesen werden. Das Unterabtasten basiert darauf, dass das menschliche Auge Farbinformation weniger genau wahrnehmen kann als Helligkeitsinformation. Daher wird die Abtastrate der Farbkomponenten Cb und Cr gegenüber der Grundhelligkeit Y reduziert, sodass mehrere Pixel zusammen nur einfach abgetastet werden, ohne dass dadurch die wahrgenommene Bildqualität abnimmt. Es existieren verschiedene Formen der Unterabtastung, bei dem zumeist verwendeten 4:2:0-Format entstehen für einen 16×16 - Makroblock ein 16×16 Y-Block sowie je ein 8×8 Cb-Block und ein 8×8 Cr-Block. Der Speicherbedarf für die beiden letzten Blöcke wird also halbiert. Je nach Einstellung werden diese Blöcke dann weiter zerlegt in 8×8 oder 4×4 Blöcke, die dann als Input für die folgende Transformation verwendet werden.

In älteren Codecs wird eine *diskrete Kosinustransformation* (DCT) angewandt, bei H.264 wird wiederum eine der DCT ähnlichen *Integertransformation* verwendet. Bei der DCT werden die Daten der einzelnen Blöcke in den Frequenzbereich transformiert, indem sie als Summe von Kosinusfunktionen dargestellt werden. So entstehen Blöcke von DCT-Koeffizienten, wobei viele Koeffizienten der höheren Frequenzen grundsätzlich sehr klein ausfallen. Diese Werte werden dann von den niedrigen hin zu den höheren Frequenzen linearisiert und man erhält eine Sequenz von Koeffizienten, bei der ab einem bestimmten Punkt alle Werte klein sind. Diese Sequenz wird dann weiter komprimiert, indem der vordere Teil verlustfrei codiert wird, zum Beispiel mit einer Huffman-Codierung, während der hintere Teil nur durch eine Lauflängencodierung angegeben wird, bei der alle Werte gleich Null sind. Die hohen Frequenzen werden also ab einem bestimmten Wert abgeschnitten und nicht gespeichert. Eine zusätzliche verlustbehaftete Komprimierung kann an dieser Stelle auch erreicht werden, indem die Quantisierung eingeschränkt wird, sodass die Anzahl an Bits pro Koeffizient reduziert wird.

Codierung mehrerer Frames

Mit dem bisher beschriebenen Verfahren können einzelne Bilder codiert werden. Videos enthalten jedoch sehr viel redundante Information, weil sich zum Beispiel Abschnitte eines Bildes von einem Frame zum nächsten gar nicht verändern, oder nur ihre Position aufgrund von Bewegungen im Bild verschoben wird. Um diese Eigenschaften auszunutzen werden Frames bei der Codierung in *I-Frames*, *P-Frames* und *B-Frames* klassifiziert. I-Frames enthalten die vollständigen Informationen, um ein Bild komplett wiederherstellen zu können. Sie werden generell bei Szenenwechseln und periodisch zwischen den anderen Frame-Typen verwendet. P-Frames

werden mithilfe der vorhergehenden Frames codiert. Dazu werden die Makroblöcke je nach Einstellung in 4×4 bis 16×16 große Partitionen zerlegt und die Partitionen nur über ihre Differenz zum vorhergehenden Frame gespeichert. Sind diese identisch, dann müssen sie gar nicht codiert werden und der entsprechende Speicherplatz wird eingespart. Zusätzlich wird mit Suchalgorithmen in vorhergehenden Frames nach ähnlichen Partitionen gesucht. Bei einem Fund wird eine Partition nur als Bewegungsvektor von ihrer ursprünglichen Position aus zusammen mit einer Differenz codiert. Bei H.264 können Bewegungsvektoren auf $1/4$ Pixel genau angegeben werden. B-Frames erweitern P-Frames noch einmal dadurch, dass sich Bewegungsvektoren neben vorhergehenden Frames zusätzlich auch auf nachfolgende Frames beziehen können. Bei der Decodierung muss dann zuerst das entsprechende nachfolgende Frame verarbeitet werden, bevor ein B-Frame decodiert werden kann. B-Frames haben nur etwa die halbe Größe eines P-Frames, haben jedoch den Nachteil, dass die nachfolgenden Frames, auf die Bezug genommen wird, zuerst decodiert werden müssen und dadurch der Speicherbedarf innerhalb eines Decoders sowie die Verarbeitungsverzögerung ansteigen.

Bei H.264 können sich P- und B-Frames auf bis zu 16 verschiedene vorhergehende Frames beziehen. Zudem können Blöcke aus verschiedenen Frames mit unterschiedlichen Gewichtungen verrechnet werden, um etwa Fading-Effekte effizient zu codieren. Eine weitere Neuerung bei H.264 ist die Intra Prediction, die auch in I-Frames zum Einsatz kommt. Dabei werden die Werte einiger Blöcke innerhalb eines Frames als die Differenz zu ihren umliegenden Blöcken codiert.

2.3.2 Video-Übertragung

Bei einem Videostream wird ein Video kontinuierlich von einem Server zu einem Client übertragen, während die beim Client empfangenen Daten sofort abgespielt werden. Beim *On-Demand-Streaming* wird ein auf einem Server gespeichertes Video auf Anfrage zu einem Client übertragen, während beim *Live-Streaming*, wie etwa bei Videotelefonaten, ein Videobild in Echtzeit aufgenommen und zum Client versendet wird. Um Videostreaming zu ermöglichen, muss zunächst eine Kommunikationssitzung zwischen Server und Client aufgebaut werden. Das Video muss dann beim Server codiert und über ein Transportprotokoll an den Client gesendet werden, wo es dann entsprechend decodiert und dargestellt wird. Zur Erläuterung dieser Vorgänge werden in diesem Abschnitt beispielhaft zunächst das *Session Initiation Protocol* (SIP) und dann das *Real-Time Transfer Protocol* (RTP) vorgestellt.

SIP

SIP [81, 82] ist ein Netzwerkprotokoll, mit dem Multimedia-Kommunikationssitzungen zwischen Teilnehmern aufgebaut und verwaltet werden können. SIP ist textbasiert, verwendet ein Anfrage-Antwort-Schema, basiert in der Regel auf UDP/IP und wird neben Videostreaming auch für Internettelefonie und Instant Messaging verwendet.

SIP-User kommunizieren über SIP-User-Agents, die über eine dem Teilnehmer zugeteilte geräteunabhängige SIP-URI angesprochen werden können. Die User-Agents registrieren sich dafür bei einem Registrar, wo ihre SIP-URI auf ihre aktuelle IP-Adresse abgebildet wird, sodass andere Teilnehmer diese beim Registrar erfragen können. Hat der User-Agent eines initiiierenden Teilnehmers so die Adresse des gewünschten anderen Teilnehmers ermittelt, sendet er eine *INVITE*-Nachricht an diesen, die dann über Proxy-Server an dessen User-Agent geleitet wird. Bei positiver Antwort wird dann eine Sitzung gestartet.

Zur Beschreibung der Sitzung, sowie zur Verhandlung der Gerätefähigkeiten der Teilnehmer, kommt bei SIP das *Session Description Protocol* (SDP) [83] zum Einsatz. SDP-Nachrichten werden dabei den SIP-Nachrichten angehängt und bestehen aus einer Liste von Beschreibungstyp-Wert-Paaren. Der im Qualitätsanpassungskontext relevanteste Eintrag ist die Medienbeschreibung. Darin werden die Art des Mediums (z.B. Audio oder Video), der Port, der zur Übertragung verwendet wird, das verwendete Transportprotokoll (z.B. RTP), sowie die unterstützten Medieneencodes deklariert. Weitere Beschreibungseinträge beinhalten etwa Senderinformationen oder Informationen über die maximale Bandbreite der Sitzung.

RTP

RTP [84] ist ein Netzwerkprotokoll zum Transport von Daten mit Echtzeitanforderungen. Das Medium wird dabei paketiert und verschickt, wobei zu den Headern der darunterliegenden Protokoll-Schichten (wie etwa UDP/IP) noch ein zusätzlicher RTP-Header hinzugefügt wird. Dieser enthält unter anderem einen Payload-Type, eine Sequenznummer, einen Zeitstempel sowie einen Identifikator des sendenden Teilnehmers. Der Payload-Type beschreibt die Codierung des Mediums und wird durch einen Code angegeben, dessen Bedeutung durch ein RTP-Profil, wie z.B. das *RTP Profile for Audio and Video Conferences* (RTP/AVP) [85], festgelegt wird. Die Sequenznummer wird verwendet, um die korrekte Reihenfolge der Pakete festzustellen, sowie Paketverluste beim Empfänger zu erkennen.

Neben RTP-Paketen werden in einer RTP-Sitzung zusätzlich Pakete des *RTP Control Protocol* (RTCP) verschickt, wodurch die Sitzung analysiert und kontrolliert werden kann. Der Großteil dieser Pakete sind sogenannte *Sender-* und *Receiver-Reports*, die in regelmäßigen Abständen vom Sender oder Empfänger des Streams verschickt werden. Bei bidirektionalen Streams, wie etwa Videotelefonie, senden beide Teilnehmer beide Arten von Reports. Die Receiver-Reports enthalten Informationen über die Anzahl verlorener Pakete, den Jitter⁸ sowie über den zuletzt erhaltenen Sender-Report. Die Sender-Reports enthalten Zeitstempel sowie die Anzahl der bisher versendeten RTP-Pakete und Bytes. Aus den Reports lassen sich dann zusätzliche Maße wie die *Round-Trip-Time* eines RTP-Pakets, die verfügbare Bandbreite und der notwendige Zwischenpuffer berechnen. Dadurch ist RTCP ein wichtiges Werkzeug zur Feststellung, ob aufgrund der Netzwerkverhältnisse eine Qualitätsanpassung des versendeten Streams benötigt wird.

2.3.3 Gründe für Qualitätsanpassungen

Wie schon erwähnt, unterliegt die Verarbeitung von Multimediadaten weichen Echtzeitanforderungen. Ein Decoder-Prozess muss zum Beispiel in der Lage sein, alle 40 ms ein neues Frame zu decodieren, um eine Framerate von 25 Frames pro Sekunde zu erzielen. Genauso müssen bei einem Videostream über das Internet die encodierten Frames rechtzeitig beim Client ankommen, damit der Decoder eine ununterbrochene Wiedergabe des Video-Signals realisieren kann. Dies hängt wiederum von der aktuellen Auslastung des Netzes ab, aber auch von der Ressourcenauslastung des Senders.

Bei einem Videoübertragungsszenario spielen somit folgende Faktoren eine entscheidende Rolle für die Einhaltung der Verarbeitungsfristen und die Erfüllung der Echtzeitanforderungen:

1. Die Codierungsgeschwindigkeit des Senders, falls der Video-Stream in Echtzeit codiert oder aus einem anderen Format transcodiert wird.
2. Die verfügbare Bandbreite der Netzwerkverbindung.
3. Die Decodierungs- und Darstellungs-Geschwindigkeit des Empfängers.

Wie bereits besprochen, stellen Übertragungs-Protokolle wie RTP und RTCP Mechanismen zur Verfügung, mit denen festgestellt werden kann, ob das Kommunikationsnetzwerk gerade einen Engpass darstellt. Vor allem bei interaktiven Live-Streams, wie bei der Videotelefonie, ist die maximal tolerierbare Verzögerung begrenzt – zu späte Pakete müssen verworfen werden und bei

⁸die mittlere Abweichung von der durchschnittlichen Verzögerung zwischen dem Senden und Empfangen eines Pakets

Paketverlusten reicht die Zeit nicht aus, um die Pakete neu zu verschicken. Wenn das Netz gerade einen Flaschenhals darstellt, unterstützen viele Streaming-Anwendungen, wie z.B. Skype, Jitsi, Youtube usw., eine dynamische Anpassung der Qualität und somit eine Reduktion der Datenrate des gesendeten Video-Signals, um dem Engpass entgegenzuwirken.

Anpassungen dieser Art sind jedoch derzeit nur anhand der verfügbaren Netzwerkbandbreite und nicht aufgrund der Auslastung der CPU-Ressourcen des Senders und des Empfängers möglich. Der Grund dafür ist einerseits, dass solche Anwendungen auf Mehrzweck-Systeme aufgebaut sind, bei denen die Ressourcenauslastung der Endsysteme nur kaum oder gar nicht erfasst und kontrolliert werden kann. Auf der anderen Seite fehlen weitere Mechanismen zur Synchronisation von Sender und Empfänger aufgrund ihrer verfügbaren Ressourcen und ihrem Verarbeitungsfortschritt.

Die Codierungs- und Decodierungsgeschwindigkeit hängt zum einen von den verfügbaren CPU-Ressourcen und zum anderen von der Komplexität der Komprimierungsalgorithmen des verwendeten Codecs, sowie der verwendeten Qualitätseinstellungen ab. Verringern sich die verfügbaren CPU-Ressourcen zu sehr, da etwa nebenläufig weitere Prozesse gestartet werden, muss eine weniger aufwendige Komprimierung gewählt werden oder die Qualität des Video Streams reduziert werden.

Mit der Integration solcher multimedialen Anwendungen innerhalb der Ressourcensteuerungs-Mechanismen des in dieser Arbeit vorgestellten Modells, können jedoch Szenarien realisiert werden, bei denen auch eine Anpassung der Qualität eines übertragenen Videos aufgrund der lokalen Ressourcenauslastung eines Senders oder eines Empfängers bewerkstelligt werden kann. Ein solches Szenario wird im Kapitel 5 für eine Video-on-Demand Anwendung vorgestellt.

Im Folgenden werden zunächst Möglichkeiten der technischen Realisierung einer dynamischen Anpassung der Qualität einer Anwendung, unter anderem am Beispiel eines übertragenen Videos, untersucht und bewertet.

2.4 Qualitätsanpassungen

Wie im Rahmen der Anforderungen im Kapitel 1.3 bereits argumentiert wurde, soll für die Überlastkontrolle innerhalb eines ARTOS-Systems der Ressourcenbedarf von Anwendungen durch eine Anpassung ihrer bereitgestellten Qualität dynamisch gesteuert werden können. Eine Möglichkeit für die Realisierung solcher Anpassungen besteht darin, Anwendungen anhand eines

entsprechenden Applikationsmodells mit zur Laufzeit umschaltbaren Betriebsmodi zu entwickeln. In jedem Betriebsmodus hat dann eine Anwendung einen unterschiedlichen Ressourcenverbrauch und erbringt einen bestimmten Nutzen im System. In dieser Arbeit wird zunächst von einer monotonen Beziehung zwischen dem Ressourcenverbrauch und der erbrachten Qualität einer Anwendung ausgegangen. Hierbei werden die Begriffe Qualität und Nutzen als Synonyme interpretiert und verwendet.

Die monotone Beziehung zwischen dem Ressourcenverbrauch und der erbrachten Qualität einer Aktivität ist in Abbildung 2.8 beispielhaft dargestellt. Ein höherer Modus m_i generiert einen höheren Nutzen U unter Voraussetzung einer größeren Ressourcenmenge R . Innerhalb des in Kapitel 3.1 definierten Applikationsmodells wird eine Anwendung als eine Menge von Aktivitäten (Tasks) mit Echtzeitanforderungen und unterschiedlichen Qualitätsstufen definiert. Der Betriebsmodus einer Anwendung etabliert sich dann als eine bestimmte Konfiguration einzelner Task-Modi. In einem laufenden ARTOS-System werden die Qualitätsstufen der Tasks also nicht einzeln ausgehandelt, sondern nach den Vorgaben der Anwendungsmodi direkt eingestellt. Somit wird auch der Ressourcenbedarf und der Nutzen einer Anwendung mit der Granularität ihrer Betriebsmodi abstrahiert. Dabei gilt dieselbe Monotonie-Bedingung wie auf der Task-Ebene – für einen höheren Nutzen steigt die kumulative Rechenzeitanforderung der gesamten Anwendung. Der hierarchische Scheduler, welcher im Kapitel 3.2 vorgestellt wird, arbeitet auf seiner obersten Ebene mit den abstrahierten Werten der Anwendungsmodi und stellt die einzelnen Konfigurationen der Tasks anhand der selektierten Qualitätsstufen der Applikationen und ihren Ressourcenverbräuchen bzw. Nutzen ein.

Abbildung 2.8: Monotone Beziehung zwischen Ressourcenverbrauch und Nutzen

Im weiteren Verlauf dieses Abschnitts werden technische Mechanismen zur dynamischen Anpassung der Qualität und somit des Ressourcenverbrauchs von Anwendungen und ihren Aktivitäten untersucht. Dabei werden im folgenden Kapitel 2.4.1 zunächst Möglichkeiten zur Realisierung von Qualitätsstufen bzw. Betriebsmodi auf der Task-Ebene diskutiert. Darüber hinaus werden

zwei Fallbeispiele, für die Verarbeitung von Video-Signalen und für die dynamische Bereitstellung von Inhalten innerhalb web-basierter Anwendungen, genauer untersucht. Wie der tatsächliche Ressourcenverbrauch einer solchen Anwendung innerhalb einer bestimmten Qualitätsstufe ausfällt und wieviel Ressourcen durch die Degradierung ihrer Aktivitäten in niedrigeren Betriebsmodi eingespart werden können, hängt vom tatsächlichen Anwendungsszenario, von der Realisierung der Qualitätsstufen und von der konkret verwendeten Ausführungsplattform ab. Solche Betrachtungen werden anhand eines ARTOS-kompatiblen VoD-Szenarios im Rahmen der Evaluation in Kapitel 5 angestellt. In diesem Kapitel geht es zunächst um die Realisierungsmöglichkeiten dynamischer Qualitätsanpassungen für eine bessere Ressourcenkontrolle.

2.4.1 Realisierung von Betriebsmodi

In diesem Abschnitt werden Möglichkeiten zur Realisierung von Betriebsmodi auf der Task-Ebene untersucht. Nach dem vorgestellten Modell in Kapitel 3.1 etabliert sich der Betriebsmodus einer Anwendung aus einer bestimmten Konfiguration einzelner Task-Modi. Aus diesem Grund sind hier Mechanismen und Modelle gesucht, mit denen die Berechnungskomplexität einzelner Anwendungsaufgaben oder Tasks dynamisch auf Kosten der Präzision oder allgemein der Qualität ihrer bereitgestellten Ergebnissen heruntergestuft werden kann.

Die Realisierung unterschiedlicher Qualitätsstufen von Tasks wurden zum Beispiel bereits von Liu et al. im Rahmen der Imprecise Computations [54, 55] behandelt. Generell wurden dort drei Methoden zur dynamischen Steuerung der von einer Task generierten Arbeitslast unterschieden: *Sieve Functions*, *Monotone Tasks* und *Multiple Versions*.

Bei den **Sieve Functions** handelt es sich um Berechnungen, die unter Umständen keine Verbesserung der Eingabe-Daten oder eines alten Ergebnisses liefern. Ein entsprechender Scheduler soll in der Lage sein dynamisch zu entscheiden, ob solche Berechnungen ausgeführt oder vollständig umgangen werden können. Da diese Entscheidung anwendungsspezifisch ist und von der Beschaffenheit der berechneten Funktion und der Eingangsdaten abhängt, sind hier ein spezielles Abstraktionsmodell und Schnittstellen zwischen Scheduler und Anwendung notwendig.

Bei den **Monotonen Tasks** wird vorausgesetzt, dass die Qualität eines Zwischenergebnisses innerhalb einer Berechnung besser wird, je länger eine Task ausgeführt wird. Dieses Modell kann für die Implementierung von *Anytime Algorithmen* [86] gewählt werden, die stets ein gültiges Ergebnis liefern, auch wenn ihre Berechnung vorzeitig abgebrochen wird. Ein Scheduler kann somit dynamisch über die einer Task zur Verfügung gestellte Rechenzeit (bzw. über ihre Ausführungsdauer) eine bestimmte Ergebnispräzision erzielen. Um von einem kontinuierlichen Eingaberaum

für den Scheduler abzusehen, kann das Modell der monotonen Tasks mit *Milestones* und *Checkpoints* erweitert werden, an denen gezielt eine bestimmte Präzision erreicht und ein berechnetes Zwischenergebnis gespeichert wird (siehe Abbildung 2.9a). Wenn eine Berechnung vorzeitig abgebrochen wird, dann gilt die letzte erreichte Qualitätsstufe m_i . Die Milestones kommen den in dieser Arbeit vorgeschlagenen Betriebsmodi der Tasks konzeptionell näher.

Abbildung 2.9: Realisierung von Task-Modi (U = Qualität, C = Kosten)

Bei den **Multiple Versions** liegt eine Task in unterschiedlichen Implementierungsvarianten vor, die dynamisch bei jedem Job umgeschaltet werden können (siehe Abbildung 2.9b). Damit kann gezielt eine bestimmte Qualität erbracht werden und es existiert zunächst keine Einschränkung für die Art des berechneten Algorithmus (z.B. Anytime). Während eine Primär-Version ein Ergebnis mit der gewünschten Präzision ausliefert, jedoch den höchsten Berechnungsaufwand besitzt, können mehrere alternative Versionen mit einer niedrigeren Komplexität und Präzision arbeiten. Im Falle eines Ressourcenengpasses kann ein Scheduler dynamisch eine der alternativen Versionen auswählen, um zum Beispiel einer Überlast im System entgegenzuwirken. Hier ist also ein dynamischer Trade-off zwischen Ressourcenverbrauch und Qualität der Tasks notwendig. Bei den Imprecise Computations wird zum Beispiel ein Scheduling-Mechanismus (NORA) vorgestellt, der einen minimalen, aufgrund von Task-Degradierungen generierten totalen Fehler im System zu bewahren versucht [55]. Berechnungen der Tasks, die über eine Grundqualität hinaus zu einem Ergebnis mit besserer Qualität führen (Optional Parts), werden nur opportunistisch innerhalb reservierter Kapazitäten ausgeführt und können bei Schwankungen der Ressourcenverbräuche jederzeit abgebrochen werden.

Das im Kapitel 3.1 vorgestellte Applikationsmodell unterstützt generell die Implementierung multipler Task-Versionen und Milestones monotoner Tasks. Der durch den hierarchischen Scheduler generierte Nutzen innerhalb eines ARTOS-System wird jedoch gezielt durch die Auswahl bestimmter Qualitätsstufen der Anwendungen und ihrer Tasks erbracht. Dabei wer-

den die verfügbaren Ressourcen unter den Anwendungen entsprechend aufgeteilt, wobei auch Schwankungen innerhalb der Ressourcenverbräuche der Tasks berücksichtigt werden. Im Kapitel 4.4.2 wird eine Realisierung unterschiedlicher Task-Versionen mittels RTSJ [27] anhand von **Interruptible**-Instanzen vorgestellt. Ein **Interruptible**-Objekt repräsentiert einen dynamisch selektierbaren und ausführbaren Code-Abschnitt innerhalb einer Anwendung. Rein-monotone Tasks ohne verschiedene Versionen können hiermit ebenfalls realisiert werden, indem bei einem Modus-Wechsel bestimmte Qualitätsparameter⁹ oder Abbruchbedingungen für die Berechnungen einer Task innerhalb desselben **Interruptible**-Objekts eingestellt werden.

Eine andere Möglichkeit für die Anpassung der Ressourcenverbräuche und der Qualität von Aktivitäten kann anhand einer dienst-basierten Architektur (*Service Oriented Architecture*) realisiert werden. Durch eine dynamische Auswahl unterschiedlicher Dienstimplementierungen zur Laufzeit können Anwendungen in verschiedenen Betriebsmodi neu-komponiert werden. Eine solche Architektur wird zum Beispiel durch OSGi [87] bereitgestellt. OSGi wurde bei der Implementierung von ARTOS unter anderem für die Realisierung einer dynamischen Installation und Aktivierung von Java-basierten Anwendungen verwendet (siehe Kapitel 4.1).

Während in diesem Kapitel Methoden für die Realisierung von Betriebsmodi diskutiert wurden, wird in den folgenden Kapiteln 2.4.2 und 2.4.3 auf konkrete technische Mechanismen zur Anpassungen der Qualität und des Ressourcenbedarfs innerhalb von zwei Anwendungsszenarien eingegangen. Dabei werden zunächst Möglichkeiten einer subjektiven sowie objektiven Einschätzung der von einer Anwendung erzeugten Qualität diskutiert.

2.4.2 Fallstudie: Qualität eines Videosignals

Die Qualität eines Videos kann entweder objektiv anhand mathematischer Modelle oder subjektiv anhand von Nutzerbefragungen bestimmt werden [79]. Objektive Kriterien liefern dabei genaue und wiederholbare Bewertungen, können jedoch nicht exakt die subjektive und situations-abhängige menschliche Wahrnehmung widerspiegeln. Ein simples objektives Maß für Bilder eines Videos ist das Spitzen-Signal-Rausch-Verhältnis (*Peak Signal to Noise Ratio* (PSNR)). Dieses wird auf einer logarithmischen Skala in Dezibel (dB) angegeben und vergleicht die *mittlere quadratische Abweichung* [88] (Mean Squared Error (MSE)) eines Frames K aus dem komprimierten Video von seinem Gegenstück O im Original-Video, mit dem Quadrat des höchstmöglichen Signalausschlags: (n ist hierbei die Anzahl von Bits pro Frame)

⁹wie z.B. der Quantisierungsparameter bei einer Video-Codierung, (siehe Kapitel 2.4.2)

$$MSE = \frac{1}{m \cdot n} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (O(i, j) - K(i, j))^2 \quad (2.11)$$

$$PSNR_{db} = 10 \log_{10} \frac{(2^n - 1)^2}{MSE} \quad (2.12)$$

Wie man in Gleichung 2.11 sieht, wird die mittlere quadratische Abweichung zweier Bilder pixel-weise berechnet. Je höher der *MSE*-Wert ausfällt, desto kleiner wird der entsprechende *PSNR*-Wert in Gleichung 2.12, also nimmt dabei die Qualität eines Vergleichs-Frames gegenüber seinem Original ab. Somit kann anhand des durchschnittlichen *PSNR*-Werts für eine Sequenz von Bildern, ein Maß für den Qualitätsunterschied zweier unterschiedlich codierten Versionen eines Videos bestimmt werden. Dieser Qualitätsvergleich wird im Rahmen der Evaluation in Kapitel 5.7 zunutze gemacht.

Eine subjektive Qualitätsmessung kann ähnlich wie beim *Mean Opinion Score* (MOS) für Video-Signale durchgeführt werden. Es werden verschiedene Videosequenzen unterschiedlicher Art (zum Beispiel ruhige Aufnahmen, sowie Aufnahmen mit vielen Bewegungen) ausgewählt und einer Anzahl von Testzuschauern vorgespielt. Diese bewerten die Videoqualität und die Bewertungen aller Testzuschauer werden dann zu einem Durchschnittswert verrechnet. Das von der ITU-T standardisierte Verfahren für „Perceptual evaluation of video quality“ (PEVQ) [89] ermöglicht eine Qualitätsabschätzung anhand tatsächlicher Qualitäts-Parameter eines Video-Signals, die mit den in unterschiedlichen subjektiven Tests ermittelten MOS-Faktoren korreliert.

Framegröße, Framerate und Framequalität

Im Kapitel 2.3 wurden die wichtigsten Grundlagen bei der Codierung von Video-Signalen vorgestellt. In diesem Abschnitt werden Möglichkeiten zur Einstellung der Qualität eines codierten Videos durch entsprechende Parametrisierung eines Encoders am Beispiel von H.264 untersucht.

Die wichtigsten Parameter, die bei der Videocodierung zur Einstellung der Qualität verwendet werden können, sind die *Framegröße*, die *Framerate* sowie die *Framequalität*. Alle drei Parameter wirken sich direkt auf die Bitrate des codierten Videos sowie auf die wahrgenommene Videoqualität aus [90].

Die Framegröße oder Bildauflösung ist festgelegt durch die Anzahl von Pixeln pro Frame und wird durch $Breite \times Höhe$ des Bilds in Pixeln angegeben. Je höher die Bildauflösung ist, desto mehr Pixel pro Frame müssen verarbeitet werden, wodurch auch der Aufwand für einen Encoder/Decoder steigt.

Die Framerate ist die Frequenz, mit der die einzelnen Frames eines Videos bereitgestellt und visualisiert werden, sie wird in *Frames per Second* (FPS) angegeben. Da ein Codec periodisch arbeitet, steht die Framerate in einem direkten Bezug zur Zeitschranke, mit welcher jedes einzelne Frame verarbeitet werden muss, um die vorgegebene Rate zu erfüllen. Eine Verringerung der Framerate bedeutet generell eine Vergrößerung der Codec-Periode und somit auch eine Relaxierung seiner Rechenzeitanforderung, bei gleichbleibender Frame-Qualität.

Die Framequalität wird beim Encoder wiederum dadurch bestimmt, wie genau die einzelnen Blöcke nach der Transformation (z.B. DCT) quantisiert werden (vgl. Kapitel 2.3). Die Quantisierungsgenauigkeit bestimmt in etwa, wie viele Koeffizienten als 0 codiert werden und wie viele Bits für die Codierung der übrigen Koeffizienten verwendet werden (siehe Kapitel 2.3). Sie wird bei Encodern häufig durch einen *Quantisierungsparameter* (QP) bestimmt. Bei H.264 kann der QP einen Wert zwischen 1 und 51 einnehmen, wobei mit $QP = 1$ eine verlustfreie Quantisierung eingestellt wird, während höhere Werte eine zunehmend verlustbehaftete Quantisierung implizieren.

Die Framegröße und Framerate lassen sich bei Encodern meist direkt einstellen und werden in der Regel über das gesamte Video konstant gehalten. Es ist möglich, diese Werte gegenüber einer Original-Aufnahme zu modifizieren. Das codierte Video wird dann vom Encoder entsprechend skaliert. Eine Erhöhung gegenüber der Original-Aufnahme ist jedoch weniger sinnvoll, da hier zusätzliche Pixel oder Frames mit redundanter Information gefüllt werden müssen. Für eine Einstellung der Framequalität existieren unterschiedliche Strategien, die mit dem Verwendungszweck des Videos zusammenhängen [91, 92]:

- **Constant QP (CQP):** Der QP wird für alle Frames konstant gehalten, und somit haben diese die gleiche Qualität. Da verschiedene Frames jedoch unterschiedlich viel Redundanz enthalten, schwankt dadurch die Bitrate und der Berechnungsaufwand ist im Verlauf des Videos nicht vorhersehbar.
- **Constant Rate Factor (CRF):** Diese Einstellung ist CQP ähnlich, es wird jedoch hier versucht, die subjektive Bildqualität konstant zu halten. Dies wird realisiert indem in Frames mit viel Bewegung der QP geringer eingestellt wird als in stillen Frames, da die menschliche Wahrnehmung stille Objekte genauer wahrnimmt. Dadurch wird eine höhere

subjektive Bildqualität erreicht als bei CQP, die Bitrate und der Berechnungs-Overhead schwanken jedoch nach wie vor unvorhersehbar.

- Constant Bit Rate (CBR): Jedes Frame bekommt die gleiche Anzahl an Bits, indem der QP jeweils entsprechend angepasst wird. Dadurch schwankt zwar die Qualität unter Umständen stark, die Bitrate und somit der relative Berechnungsaufwand bleiben jedoch konstant.
- Average Bit Rate (ABR): Bei dieser Einstellung wird versucht, eine durchschnittliche Bitrate einzuhalten, indem sowohl die Bitmenge als auch der QP von einem Frame zum nächsten entsprechend variiert wird. Damit der Encoder nicht „raten“ muss, welche Bitrate für Folgeframes benötigt, wird häufig ein *2-Pass Encoding* durchgeführt. Dabei durchläuft der Encoder einen Teil, oder das ganze Video zunächst einmal, ohne einen Output zu generieren und lernt dabei, wie die Bitrate am besten auf die Frames verteilt werden kann.

Für ein Video-Streaming Szenario über das Internet bietet sich eine relativ konstante Bitrate wie durch CBR und ABR an, da man dynamische Bitraten-Peaks vermeiden möchte, weil es sonst zu Paketverlusten kommen könnte. Auf der anderen Seite bleibt mit CBR auch der Codierungs- und Decodierungsaufwand auf Kosten einer dynamischen Video-Qualität relativ konstant. Möchte man zum Beispiel Schwankungen im Ressourcenverbrauch eines Codecs vermeiden ist diese Strategie besser.

Internet Streaming Dienste bieten Videos in der Regel in unterschiedlichen Framegrößen an. Meist wird dabei nur die Höhe angegeben, da die Breite je nach Seitenverhältnis des Videos entsprechend unterschiedlich ausfällt. Häufige Formate sind 144p, 240p, 360p, 480p, 720p („HD“) und 1080p („Full HD“). Grundsätzlich macht es keinen Sinn, eine Framegröße zu wählen, die größer ist als der Abspiel-Bildschirm. Dabei wird die wahrgenommene Qualität nicht erhöht, aber die Bitrate ist höher und es wird zusätzliche Rechenleistung benötigt, um das Video zu decodieren und es auf die richtige Auflösung zu skalieren. Um Engpässe innerhalb der verfügbaren Netzwerkbandbreite oder CPU-Ressourcen zu beheben, kann die Framegröße geringer als der Abspiel-Bildschirm gewählt werden. Dabei nimmt jedoch die wahrgenommene Qualität signifikant ab. Nichtsdestotrotz ist zusätzliche Prozessorleistung notwendig, wenn das Bild auf die gesamte Bildschirmgröße hochskaliert werden soll [93].

Die in Film und Fernsehen am meisten verwendeten Frameraten sind 24, 25 und 30 FPS. Eine Reduktion der Framerate reduziert auch die Bitrate und die Prozessorauslastung beim Codieren / Decodieren, die Wirkung ist jedoch nicht linear mit der Reduktion. Vor allem bei Frames mit viel Bewegung nimmt die Ähnlichkeit zwischen einzelnen Frames ab, und so wird die mögliche Komprimierung durch Bewegungskompensation reduziert [93].

Eine Reduktion der Framequalität reduziert ebenfalls die Bitrate sowie die Prozessorauslastung beim Codieren und Decodieren. Da die einzelnen Blöcke mit weniger Bits codiert werden, wird die Komprimierung durch Bewegungskompensation weniger aufwendig.

Die Studien von Buchinger et al. [94] und von McCarthy et al. [95] haben gezeigt, dass Benutzer in bestimmten Situationen eine Qualitätsanpassung durch Reduktion der Framerate einer Reduktion der Framequalität vorziehen, vor allem wenn es sich um Videosequenzen mit viel Bewegung handelt. Brandt und Wolf [93] haben in einer subjektiven Qualitätsstudie befunden, dass für eine optimale Qualitätsanpassung zunächst die Framegröße der Größe des Abspielbildschirms angepasst werden sollte, dann die Framerate reduziert und erst zuletzt, falls nötig, die Framequalität angepasst werden sollte, um die gewünschte Bitrate zu erreichen.

Profile und Level

Bei H.264 kann mit *Profilen* [80] angegeben werden, welche Fähigkeiten ein Gerät benötigt, um ein Video decodieren und abspielen zu können. Höhere Profile unterstützen aufwendigere Funktionen des H.264 Standards, Videos mit diesen Profilen können dann aber von einigen älteren Geräten nicht abgespielt werden. Verbreitete Profile sind etwa das Baseline-, das Main- und das High-Profil. Beim Baseline-Profil werden beispielsweise die am meisten CPU-aufwendigen Features wie B-Frames und Weighted Prediction ausgelassen.

Neben Profilen können durch *Level* [80] weitere Kompatibilitätseinschränkungen angegeben werden. Die unterschiedlichen Level machen Angaben über die maximale Decodiergeschwindigkeit, die maximale bearbeitbare Framegröße und die maximale Bitrate. In einem Streaming-Video Szenario kann die Verwendung von Profilen zur dynamischen Qualitätsanpassung verwendet werden. Wenn sich die verfügbaren Ressourcen eines Endgeräts ändern, kann durch die Auswahl eines anderen Profils oder Levels dem Sender mitgeteilt werden, eine entsprechende Anpassung des gestreamten Videos vorzunehmen.

Weitere Encoding-Einstellungen bei H.264

Durch weitere Einstellungen für einen H.264 Encoder kann der Codier- bzw. Decodieraufwand beeinflusst und theoretisch bei Ressourcenengpässen entsprechend angepasst werden [91].

- Motion-Estimation Search Pattern und Range: Mit diesen Einstellungen wird das Suchmuster und die Suchreichweite angegeben, mit der beim Codieren nach Bewegungsvektoren gesucht wird. Sie haben somit eine große Auswirkung auf den Codierungsaufwand.

- Maximale Anzahl von Referenzframes: Diese Einstellung gibt an, auf wie viele verschiedene Referenzframes sich die Bewegungsvektoren in P- und B-Frames beziehen können. Eine hohe Einstellung erhöht den Codierungsaufwand und verlangsamt das Decodieren deutlich.

Hierarchische Modulation / Scalable Video Coding

Bei der Hierarchischen Modulation wird ein hochqualitativer Bitstrom erzeugt, der einen oder mehrere Teilströme niedrigerer Qualität enthält. Diese können extrahiert werden, indem bestimmte Pakete des Bitstroms ausgelassen werden. Diese Technik wird bei der *Scalable Video Coding Extension* für H.264 (H.264/SVC) [96] verwendet. Sowohl der vollständige als auch die Teil-Bitströme können von einem Decoder decodiert und dargestellt werden. Ein wichtiger Aspekt dabei ist, dass die skalierbaren Bitströme im Vergleich zu einem normalen Video-Strom mit gleicher Bitrate keine deutlich schlechtere Qualität haben und nicht aufwendiger zu decodieren sind. Man spricht auch vom *Base-Layer* als den minimalen Bitstrom und von *Enhancement-Layern*, die einen Bitstrom zur nächsten Skalierungsstufe erweitern.

SVC ist vor allem dann interessant, wenn ein Videostream-Anbieter mehrere Clients bedient, die über unterschiedliche Fähigkeiten und Ressourcen verfügen und damit verschiedene Anforderungen an Framegröße, Framerate und Framequalität haben. Das Video muss dann nur einmal in hoher Qualität skalierbar codiert werden. Jeder Client kann dann seinen Anforderungen entsprechend nur einen Teil des Stroms decodieren. Im Rahmen der Qualitätsanpassung für begrenzte CPU-Ressourcen kann der zu decodierende Teil des Bitstroms bei einem Client dynamisch ausgewählt werden, ohne dass eine Anpassung des gesendeten Videostroms auf der Serverseite stattfinden muss.

Bei SVC wird zwischen *temporaler Skalierbarkeit* (Skalierung der Framerate), *räumliche Skalierbarkeit* (Skalierung der Framegröße) und *Qualitätsskalierbarkeit* (Skalierung der Framequalität) unterschieden. Die verschiedenen Skalierungsmechanismen können auch kombiniert werden (*kombinierte Skalierbarkeit*).

Temporale Skalierbarkeit kann realisiert werden, indem jedem Frame ein Layer-Identifizier T zugewiesen wird, der beim Base-Layer mit 0 beginnt und bei jedem aufsteigendem Layer um 1 erhöht wird. Eine Einschränkung hierbei ist, dass jedes Frame nur solche Frames als Referenzframes für die Bewegungskompensation verwenden kann, deren Layer-Identifizier kleiner gleich dem eigenen ist. Will man dann ein Video bis zum Layer k decodieren, werden alle Frames mit $T > k$ ignoriert.

Bei der räumlichen Skalierbarkeit entspricht jeder Layer einer unterstützten Bildauflösung. Innerhalb der einzelnen Layer werden gewöhnliche Bewegungskompensation und Intra Prediction eingesetzt. Zusätzlich dazu wird dann *Inter Layer Prediction* angewendet, bei der möglichst viele Informationen der niedrigeren Layer weiterverwendet werden, um die Bitrate der einzelnen Layer zu reduzieren.

Die Qualitätsskalierbarkeit kann als Spezialfall der räumlichen Skalierbarkeit betrachtet werden, wobei bei aufsteigendem Layer nicht die Bildauflösung zunimmt, sondern die Quantisierungsgenauigkeit erhöht wird.

2.4.3 Fallstudie: Dynamische Auslieferung von Inhalten

In diesem Abschnitt werden Qualitätsanpassungen im Kontext von Web-Anwendungen untersucht. Diese Anwendungen arbeiten auf der Basis eines Anfrage-Antwort-Modells, bei dem Benutzer bestimmte dynamisch von einem Webserver mit Inhalten gefüllte und bereitgestellte Webseiten anfordern. Dabei werden Möglichkeiten zur Ressourcenkontrolle innerhalb der Webserver untersucht, indem die Bearbeitungskomplexität der Anfragen dynamisch auf Kosten der bereitgestellten Menge an Informationen reguliert werden kann. Solche Mechanismen kommen dann zum Tragen, wenn viele Nutzer gleichzeitig Anfragen starten, sodass ein entsprechender Webserver mit begrenzter Ressourcenkapazität überlastet wird und die Anfragen gar nicht oder nur sehr langsam bedienen kann. Dabei können Erwartungen für eine maximale Antwortzeit der Anfragen verletzt werden, die zu einer schlechten Nutzer-Erfahrung und somit auch zu monetären Verlusten des Seiten-Betreibers führen können. Obwohl Qualitätsanpassungen bei einer dynamischen Auslieferung von Inhalten noch nicht soweit erforscht und standardisiert wurden, wie das zum Beispiel bei der Verarbeitung von Videosignalen der Fall ist, existieren einige Beispiele, die im Folgenden grob vorgestellt werden. Solche Szenarien lassen sich einwandfrei in das innerhalb dieser Arbeit vorgeschlagene Applikationsmodell und der entsprechenden Ressourcensteuerung integrieren.

Anpassung von Web-Inhalten

Abdelzaher et al. schlagen in [97] Mechanismen zur Anpassung von Web-Inhalten vor, mit dem Ziel die Ressourcenanforderungen für die Auslieferung von Inhalten innerhalb von Webservern bei Bedarf abzusenken. Innerhalb einer Studie belegen die Autoren, dass für einen Web-Server mit einer 100 MB Netzwerkanbindung der typische Engpass bei Überlast nicht das Netzwerk sondern das Endsystem an sich ist. Die Idee ist, die Webinhalte so anzupassen, dass essentielle

Inhalte erhalten bleiben, während die Anforderungen für die Bereitstellung und Auslieferung der Inhalte abgesenkt werden.

Die grundlegende Idee ist, dass Web-Entwickler Inhalte mit unterschiedlichen *Adaptation Tags*, wie zum Beispiel *expendable*, *degradable* oder *important*, annotieren können, die dann durch Management-Werkzeuge als verschiedene Versionen einer Seite aufbereitet werden. Hier wird eine Vorverarbeitung der Inhalte vorausgesetzt, da für ein überlastetes Endsystem zusätzliche Berechnungen für Datenfilterung und Komprimierung den Server nur zusätzlich belasten würden. Als Techniken für die Degradierung der Inhalte werden *verlustbehaftete Kompression* von Bildern, *Reduktion eingebetteter Objekte* innerhalb einer Seite und *Reduktion lokaler Links* vorgeschlagen.

In einer Studie wurde ermittelt, dass bei 80 Shopping-Seiten GIF und JPG Bilder allein mehr als 65 % der übermittelten Inhalte konstituieren. In vielen Fällen können diese Bilder erheblich komprimiert werden ohne einen bemerkenswerten Qualitätsabfall (z.B. ein 74 KB großes GIF auf 8.4 KB großes JPG). Diese Kompression verringert die Übertragungszeit und spart somit Rechenzeit innerhalb des Servers. Der Qualitätsunterschied ist dabei auf den meisten Anzeigegeräten nicht sonderlich signifikant.

Von der Server-Perspektive ist die Dokumentgröße nicht so ausschlaggebend wie die Anzahl der im Dokument eingebetteten Objekte. Bei der Anforderung einer URL initiiert die Client-Seite mehrere unabhängige Anfragen um diese Objekte abzurufen. Jeder diese Server-Anfragen resultiert in einem relativ hohen fixen Overhead, zusätzlich zum variablen und von der Dokumentgröße abhängigen Aufwand. Für einen Apache Server auf HP-UX wurde eine fixe Verarbeitungszeit von 1.6 ms pro Anfrage ermittelt, plus zusätzliche 65 μ s per übermittelter Daten-Byte. Aus diesem Grund können für ein überlastetes System Verarbeitungskosten von Seitenanfragen erheblich durch Eliminierung kosmetischer Elemente (wie Icons, Bullets, Bars usw.) abgesenkt werden.

Eine andere Möglichkeit stellt die Reduktion lokaler Links dar. Diese Reduktion beeinflusst das Browsing-Verhalten von Nutzern, da weniger Zugriffsmöglichkeiten auf den Server bereitgestellt werden. Die Reduktion der lokalen Links kann, durch eine Begrenzung der Tiefe des Inhaltsbaums einer Seite ausgehend von einer Oberseite, automatisiert stattfinden. Inhalt-Provider können darüber hinaus mit speziellen Tags Subbäume indizieren, die über die spezifizierte Suchtiefe auf jeden Fall bereitgestellt werden sollen.

Durch diese Mechanismen zeigen die Autoren erhebliche Einsparungen der Rechenzeit von Anfragen in Überlastsituationen.

Partielle Ausführungen innerhalb eines Web Servers

Kim et al. präsentieren in [98] einen regelungstheoretischen Ansatz *QACO*, welcher dynamisch justierbare Ressourcenkontingente *Quotas* an einer Menge von ausstehenden Anfragen verteilt. Hier wird von sog. *partiellen Ausführungen* von Anfragen ausgegangen, bei denen Inhalte ähnlich wie beim vorherigen Ansatz von Abdelzaher et al. in unterschiedlichen Versionen vorliegen, die dynamisch ausgewählt werden können. Darüber hinaus können Suchanfragen partiell ausgeführt werden, indem eine Suchmaschine, der weniger Ressourcen zur Verfügung stehen, weniger Webseiten qualifiziert und einstuft.

Auf der Basis dieser Adaptionstechniken wird eine Regelung aufgebaut, die einen Sollwert für die Antwortzeit von Anfragen observieren soll. Somit entsteht eine optimale Konfiguration von Verarbeitungsstufen unterschiedlicher Anfragen, mit der die Einhaltung eines *Service Level Agreements* (SLA) für die Antwortzeit einer jeden Anfrage observiert werden kann. Auf diese Weise sollen die Kosten, zum Beispiel für einen Cloud-Provider, minimiert werden, die dadurch entstehen, dass in Überlastsituationen einzelner oder mehrerer Rechenknoten, ohne entsprechende Anpassungs-Mechanismen die Antwortzeit-SLA für einen oder mehreren Ressourcen-Mieter (z.B. Webseiten-Provider) weitaus stärker verletzt werden kann. Für ein solches Szenario sind natürlich entsprechende Kostenmodelle notwendig, die die Auslieferung von degradierten Inhalten entsprechend in Betracht ziehen.

Bei diesem Ansatz sind die Regelschritte der Steuerung ausschliesslich auf einer durchschnittlichen Eingabevariable über die gemessenen Ausführungszeiten von Anfragen basiert. Darüber hinaus werden weder Fristen für die Verarbeitungen einzelner Anfragen definiert und in die Regelung mit einbezogen, noch unterschiedliche Arten von Jobs und Schwankungen in ihren Ausführungszeiten in Betracht gezogen. Das Ergebnis ist, dass nachdem sich Schwankungen in den Verarbeitungszeiten der Anfragen gegenseitig kompensieren können, dieser Algorithmus keine strikte, sondern nur eine durchschnittliche Einhaltung der Antwortzeit-SLA garantieren kann.

Im Rahmen dieser Promotionsarbeit wurde bereits für ein ähnliches Cloud-basiertes Szenario ein Mechanismus [1] entwickelt, der durch das Ausnutzen von Scheduling-Mechanismen aus der Echtzeittheorie in der Lage ist, eine strikte Vorhersage der Planbarkeit von Anfragen mit unterschiedlichen Verarbeitungsqualitäten und somit über die Einhaltung einer Antwortzeit-SLA zu treffen. Dabei wurde die Anfrage-Warteschlange innerhalb eines Web-Servers als Prozess-Liste eines Schedulers, die einzelnen Anfragen als aperiodische multimodale Tasks und die durch die SLA vorgegebene Antwortzeit als Job-Fristen modelliert. Über eine Planbarkeitsanalyse durch

Buttazzo's *instantaneous load* Modell [12], kann beim Eintreffen neuer Anfragen anhand des aktuellen Zustands der Anfrage-Warteschlange und der geschätzten Ausführungszeiten eine Aussage über die Verletzung der Job-Fristen und somit der SLA gemacht werden. Wenn die instantane Last 100 % übersteigt, wird ein Degradierungsverfahren der einzelnen Anfragen angewendet, sodass eine möglichst optimale Verarbeitungsqualität eingestellt wird. Im schlimmsten Fall werden jedoch alle Anfragen mit der niedrigsten Qualität bedient, um Zeit für eine horizontale Skalierung der Anwendung innerhalb der Cloud-Plattform zu schöpfen. Die Degradierung wird zusätzlich durch ein entsprechendes Kostenmodell unterstützt, wobei Anfragen bestimmter Endnutzer bevorzugt behandelt werden können.

In Brownout [99] schlagen Klein et al. ebenfalls einen regelungstheoretischen Ansatz zur Degradierung „saturierter“ Anwendungen innerhalb einer Cloud-Plattform vor. Brownout ist ein Begriff, der aus dem Kontext von Elektrizitätsversorgungsnetzen stammt und beschreibt den Zustand eines absichtlichen Spannungsabfalls, welcher in Notfällen zur Lastminderung eingeleitet wird, um einen kompletten Stromausfall zu verhindern. In ähnlicher Weise sollen Cloud-Anwendungen das Nutzererlebnis herunterskalieren, um eine Saturierung zu verhindern. Als Beispiel dienen wieder Online-Shops, die mit einem Empfehlungssystem für ähnliche Produkte ausgestattet sind. Einer in dieser Arbeit zitierten Studie zufolge, erhöhte sich der Online-Verkauf von Songs, wenn eine Gruppe von Nutzern Empfehlungen ausgesetzt war. Eine entsprechende *Recommendation Engine* impliziert jedoch einen zusätzlichen Ressourcenverbrauch. Durch eine selektive Aktivierung solcher optionaler Komponenten, kann die benötigte Ressourcenkapazität einer Anwendung auf Kosten des Nutzererlebnisses degradiert werden, ohne ihre Grundfunktionalität zu beeinträchtigen. Dafür schlagen die Autoren ein spezielles Applikationsmodell mit einem *Dimmer*-Parameter als Steuervariable, welcher die Anzahl bereitgestellter Produktempfehlungen auf einer Web-Seite bestimmt und einen Regler, der eine bestimmte Verarbeitungsdauer (Antwortzeit) von Anfragen als Sollwert ansteuert. Die Autoren zeigen das erwartete Ergebnis, dass während ein langsamer Regler die vorgegebene Antwortzeit der Anfragen besser einhält, ein System sich nur langsam von Lastspitzen erholt.

Der Brownout Ansatz ist konzeptionell reziprok zum vorherigen Modell von QACO. Während bei QACO ein Regler Ressourcen-Quotas für Anfragen steuert aus denen sich eine bestimmte Qualität erschließt, wird bei Brownout die Anzahl der ausgeführten optionalen Anwendungsteilen dynamisch gesteuert, um eine vorgegebene Verarbeitungszeit zu erzielen.

3

Modell für Ressourcensteuerung

Wie bereits in Kapitel 1.1 beschrieben, soll eine Ressourcensteuerung für ein System entwickelt werden, das mit einer variablen Menge von Applikationen mit unbekannten und zur Laufzeit schwankenden Ressourcenanforderungen umgehen soll. Darüber hinaus können die Applikationen auch Echtzeitanforderungen haben, die bei der Ressourcenvergabe in Betracht gezogen und erfüllt werden müssen. Die Steuerung soll durch ein spezielles Applikationsmodell und einen hierarchischen Scheduler für Applikationsaktivitäten mit unterschiedlichen Qualitätsmodi unterstützt und realisiert werden. Mittels einer dynamischen Kontrolle über die von den Anwendungen erbrachte Qualität, soll auch ihr Ressourcenbedarf so angepasst werden, dass in einer konkreten Plattform der höchst-mögliche Nutzen aus einer bestehenden Menge von Ressourcen, unter Einhaltung der Echtzeitanforderungen der Anwendungen, erzielt wird. Dieser äußert sich als eine optimale Konfiguration von Anwendungsmodi, die zur Laufzeit anhand des gemessenen Ressourcenbedarfs der laufenden Anwendungen dynamisch ermittelt werden soll. Eine bereits bestehende Konfiguration soll auch möglichst lang erhalten werden, um den Rekonfigurationsaufwand der Anwendungen in unterschiedlichen Modi zu minimieren (siehe Anforderungen in Kapitel 1.3). Hierfür muss zunächst ein verstellbares Maß für tolerierbare Schwankungen im dynamisch erfassten Ressourcenverbrauch der Anwendungsaktivitäten geschaffen werden. Anhand dessen soll der Scheduler in der Lage sein die Aktivitäten zur Laufzeit so einzuplanen, dass

sie mit genügend Ressourcen für die Erfüllung ihrer Echtzeit-Anforderungen in entsprechenden Qualitätsstufen bedient werden können.

Zur Realisierung einer dynamischen Qualitätsanpassung für Anwendungen wird in dieser Arbeit ein spezielles Entwicklungsmodell für Applikationen und ihren Tasks mit mehreren Betriebsmodi vorgeschlagen, welches im folgenden Kapitel 3.1 vorgestellt und näher erläutert wird. Dieses Modell bildet auch automatisch die Echtzeitanforderungen der Applikationsaktivitäten ab, durch spezielle Definitionen für periodische Tasks mit unterschiedlichen Perioden, Fristen und zur Laufzeit gemessenen und geschätzten Kosten. Das Modell wird auf der Implementierungsebene durch eine spezialisierte Schnittstellen-Beschreibung (API) bereitgestellt, welche kompatible Anwendungen automatisch in die Analyse- und Ressourcenmanagement-Mechanismen eines laufenden ARTOS-Systems integriert (siehe Kapitel 4).

Auf der Basis dieses Modells wird in den folgenden Unterkapiteln näher auf die Messung und die Approximation des Anwendungsverhaltens zur Laufzeit eingegangen. Durch die Realisierung einer gezielten Qualitätsanpassung für die Anwendungen und ihre Tasks, erschwert sich jedoch die für die Einhaltung ihrer Echtzeitanforderungen notwendige Planbarkeitsanalyse in einem laufenden System. Dieses Problem wird abschließend gemeinsam mit einer Lösungsidee vorgestellt, welche im Rahmen dieser Arbeit konzeptionell entwickelt und realisiert wurde. Die Idee ist, den Aufwand für die Berechnung der Planbarkeitsanalyse in einem laufenden System zeitlich zu verteilen und nur für Anwendungen zu betreiben, deren Kosten-Approximation nicht mehr akkurat ist. Solange jedoch die Prognosen der Ressourcenverbräuche der Anwendungs-Tasks zutreffend sind, findet keine Neuplanung statt und die letzte eingestellte optimale Konfiguration von Anwendungs- und Task-Qualitäten bleibt bestehen. Dieser Ansatz wurde bereits in mehreren wissenschaftlichen Arbeiten [17, 15, 100] publiziert und der internationalen Echtzeit-Gemeinde vorgestellt.

Die Realisierung dieser Lösung basiert auf ein hierarchisches Scheduling-Modell, das später in Kapitel 3.2 präsentiert wird. Dieses Modell ermöglicht eine dynamische Ressourcensteuerung und Einhaltung von Echtzeitgarantien für kompatible ARTOS-Anwendungen mit unterschiedlichen Qualitätsstufen. Neben der Auswahl einer optimalen Konfiguration der Applikationsmodi überprüft der Scheduler auch zu bestimmten Zeitpunkten die Planbarkeit der konfigurierten Taskmenge. Dabei werden durch Annahmen über das aktuelle Verhalten der Anwendungsaktivitäten Kostenschätzungen etabliert, die in bestimmten Abständen mit den tatsächlichen Ressourcenanforderungen verglichen werden. Anhand dieser Schätzungen wird die benötigte Ressourcenmenge pro Anwendung bemessen und mit den verfügbaren Ressourcen im System ausgehandelt. Weichen die Schätzungen von den realen Verbräuchen der Tasks ab, dann werden sie aktualisiert und die Planbarkeit der Aktivitäten innerhalb der Ressourcenkapazitäten der betroffenen

Anwendungen überprüft. Ist diese nicht mehr gegeben, dann wird die benötigte Ressourcenmenge der jeweiligen Anwendung ermittelt und notfalls eine neue optimale Konfiguration der Anwendungsmodi berechnet.

3.1 Applikationsmodell

Abbildung 3.1: Applikationsmodell

Das Applikationsmodell von ARTOS impliziert eine dynamische Menge von n konkurrierenden Anwendungen A_i mit $i \in \{1..n\}$. Für jede Applikation A_i existieren unterschiedliche Betriebsmodi $M_{i,j} \in \mathcal{M}_i$ mit $j \in \{1..m_i\}$, die dynamisch zur Laufzeit umgeschaltet werden können. Jede Applikation besteht ferner aus einer Menge von k multimodalen Tasks $\tau_{i,k} \in \mathcal{T}_i$, die in unterschiedlichen Versionen vorliegen können (vgl. Abbildung 3.1). Im vorgestellten Modell werden zunächst nur periodische Tasks¹ angenommen. Jede Task wird also zunächst wie in einem klassischen Echtzeit-System durch eine Periode $T_{i,k,l}$, eine relative Frist $D_{i,k,l}$ und Ausführungskosten $C_{i,k,l}$ definiert.

Wie bereits in Kapitel 2.1 erklärt, wird bei der periodischen Ausführung einer Task eine endliche bzw. unendliche Menge von Aktivierungsinstanzen (*Jobs*) erzeugt, die mit einem definierten Zeitabstand $T_{i,k,l}$ eintreffen. Jeder Job besitzt somit eine absolute Aktivierungszeit $r_{i,k,t}$, auch *Release Time* genannt. Sind die Release Times aller Tasks in einem System bekannt, so spricht man von einer konkreten Task-Menge. In einem ARTOS-System wird von einer unkonkreten

¹aperiodische Tasks können durch Kapselung in Servern (wie z.B. Deferrable Server [34]) bedient werden, wodurch ihre Ausführung innerhalb eines periodischen Systems sequentialisiert werden kann

Task-Menge ausgegangen, da hier Applikationen zu beliebigen Zeitpunkten installiert und gestartet werden können. Aus diesem Grund sind Release-Zeiten der Anwendungsaktivitäten im Task-Modell zunächst nicht inbegriffen.

Jede Task besitzt eine relative Frist (*Deadline*) $D_{i,k,l}$, die sich während ihrer Ausführung als absolute Frist für jeden Task-Job etabliert. Somit besitzt der n -te Job eine absolute Frist, welche sich aus der Release-Zeit $r_{i,k,n} + D_{i,k,l}$ ergibt. Jeder aktive Job muss mit seiner Berechnung vor seiner absoluten Frist fertig werden und gegebenenfalls ein Ergebnis ausliefern. Andernfalls spricht man von einer Fristverletzung (*Deadline Miss*). Im ARTOS-Modell werden Fristverletzungen generell toleriert, es werden also nur Anwendungen mit weichen Echtzeitanforderungen unterstützt.

In einem klassischen Echtzeitsystem beziehen sich die Kosten $C_{i,k,l}$ einer periodischen Task auf die schlimmsten (WCET) oder die mittleren (ACET) Ausführungszeiten ihrer Jobs. Da die tatsächlichen Ausführungszeiten einer Task im vorgestellten Modell zunächst unbekannt sind, werden sie in einem ARTOS-System zur Laufzeit gemessen. Anhand der Messungen wird dynamisch eine Schätzung für ihre durchschnittlichen Kosten $C_{i,k,l}$ etabliert. Da die Ausführungszeiten der Tasks aufgrund von plattformspezifischen Faktoren und der Beschaffenheit von den zu berechnenden Daten schwanken können, beinhalten ihre Kostenschätzungen dynamische Ressourcenpuffer für den Jitter ihrer periodisch gemessenen Kosten. Diese Puffer werden bei ARTOS in definierten Zeitabständen anhand der aktuell gemessenen Ausführungszeiten der Task-Jobs überprüft und notfalls aktualisiert, wodurch neue Schätzungen entstehen können. Die genaue Vorgehensweise wird später im Kapitel 3.1.3 näher vorgestellt.

Task-Modi

Eine Task $\tau_{i,k}$ kann ebenfalls unterschiedliche Betriebsmodi $m_{i,k,1}, m_{i,k,2}, \dots, m_{i,k,l} \in \Lambda_{i,k}$ besitzen². Konzepte für die Realisierung von Task-Modi wurden im Kapitel 2.4.1 bereits vorgestellt. Tatsächliche Implementierungsmechanismen werden im Kapitel 4.4.2 auf der Basis von RTSJ präsentiert. Neben einer unterschiedlichen Funktion können die Modi auch verschiedene Konfigurationen für die Periode $T_{i,k,l}$ und die Frist $D_{i,k,l}$ der jeweiligen Task vorgeben. Die Kostenschätzung $C_{i,k,l}$ ist ebenfalls modus-spezifisch und sie wird dynamisch für den aktuell aktiven Modus der Task ermittelt. Somit wird also ein *Task-Modus* durch das Tripel

$$m_{i,k,l} \in \Lambda_{i,k} = (T_{i,k,l}, C_{i,k,l}, D_{i,k,l}) \quad (3.1)$$

²aus Übersichtsgründen sind die Indizes der Task-Modi $m_{i,k,l}$ in Abb. 3.1 vereinfacht

definiert, wobei $\Lambda_{i,k}$ die Menge der Modi einer Task $\tau_{i,k}$ beschreibt. Auf der Basis dieser Definition werden die Echtzeitanforderungen der Aktivitäten erfasst und später für die Planbarkeitsanalyse verwendet.

Applikationsmodi

Abbildung 3.2: Beispiel eines Applikationsmodus $M_{i,j}$

Ein Applikationsmodus $M_{i,j}$ determiniert sich durch eine konkrete *Kombination* von Task-Modi (siehe Abbildung 3.2)

$$\Lambda_{i,j}^* = \{m_{i,1,l}, m_{i,2,m}, \dots, m_{i,k,n}\} \in \mathcal{M}_i \quad (3.2)$$

und einer kumulativen Rechenzeitanforderung $R_{i,j}$ der Task-Menge $\mathcal{T}_i$. Eine Variation der Task-Menge in den unterschiedlichen Anwendungsmodi ist durch das Modell zunächst nicht vorgesehen, kann aber durch eine dynamische Deaktivierung bestimmter Tasks emuliert werden (siehe später Null-Modi). Die Rechenzeitanforderung $R_{i,j}$ eines Applikationsmodus wird im Rahmen einer lokalen Planbarkeitsanalyse für die entsprechende Applikation A_i ermittelt, welche später im Kapitel 3.2.2 genauer erläutert wird.

Ein höherer Betriebsmodus einer Anwendung besitzt generell eine höhere Anforderung, liefert jedoch eine bessere Qualität oder Nutzen (siehe Kapitel 2.4)

$$U_{i,j} = u_i(M_{i,j})/u_i(M_{i,|\mathcal{M}_i|}) \quad (3.3)$$

z.B. eine bessere Videoauflösung. Die Qualität wird formalisiert durch anwendungsspezifische Utility-Funktionen $u_i(M_{i,j})$, die durch die Anwendungsentwickler a priori bereitgestellt werden. Sie wird in einem ARTOS-System normiert als das Verhältnis des Qualitätsgewinns eines jeden Modus im Vergleich zum höchsten Modus $M_{i,|\mathcal{M}_i|}$ der jeweiligen Anwendung A_i (siehe Gleichung 3.3).

Somit ist ein *Anwendungsmodus* vollständig definiert durch

$$M_{i,j} \in \mathcal{M}_i = (\Lambda_{i,j}^*, R_{i,j}, U_{i,j}) \quad (3.4)$$

also durch eine Kombination von Task-Modi $\Lambda_{i,j}^*$, deren kumulative Rechenzeitanforderung $R_{i,j}$ und einem modusspezifischen Nutzen $U_{i,j}$.

Abbildung 3.3: Null-Modus einer Applikation

Jede Applikation A_i besitzt ferner einen sog. *Null-Modus* $M_{i,0}$ mit $R_{i,0} = u_i(M_{i,0}) = U_{i,0} = 0$, also mit einer Rechenzeitanforderung und einem Nutzen von Null, in dem die Anwendung aktuell gestoppt ist. Dabei wird jede Task $\tau_{i,k}$ in einem entsprechenden Task-Modus $m_{i,j,0}$ temporär deaktiviert (vgl. Abbildung 3.3). Ein solcher Modus garantiert die Existenz einer optimalen Ressourcenverteilung im Gesamtsystem (siehe Kapitel 3.2.3) und dient der Realisierung einer dynamischen Admissionskontrolle der Anwendungen.

Abschließend, können die Anwendungen auch mit Prioritäten a_i gewichtet werden, sodass höher priorisierte Anwendungen tendenziell mehr Ressourcen bekommen und in höheren Betriebsmodi aktiviert werden als niedriger priorisierte Applikationen. Die letztendliche Entscheidung darüber findet auf einer höheren Ebene des Ressourcenmanagement-Modells statt, was später im Kapitel 3.2.3 genauer erklärt wird.

3.1.1 Bestimmung der mittleren Task-Kosten

Wie bereits im Kapitel 2.1 erklärt, arbeiten weiche Echtzeitsysteme mit mittleren Werten von Task-Ausführungszeiten (ACET). Sind solche Werte für Anwendungsaktivitäten nicht verfügbar, oder wie in diesem Fall zwischen unterschiedlichen Ausführungsplattformen nicht übertragbar, müssen sie zur Laufzeit durch Messungen ermittelt werden. Anhand dessen muss dann eine Ressourcensteuerung dynamisch die benötigte Ressourcenmenge für eine fehlerfreie Ausführung der Anwendungen und ihrer Aktivitäten ermitteln und bereitstellen.

Da das ARTOS-Modell nur periodische Aktivitäten erlaubt, wobei aperiodische Aktivitäten wie bereits erwähnt in periodischen Servern gekapselt werden könnten, werden zur Ermittlung der ACET-Werten zunächst die „reinen“ CPU-Kosten einer jeden Task pro Task-Periode wie in Abbildung 3.4 gemessen. Die reinen CPU-Kosten stellen die Rechenzeit dar, die eine Task tatsächlich auf einer Recheneinheit bekommen hat und dort ausgeführt wurde. Diese Zeiten werden zu Beginn und am Ende eines jeden Task-Jobs ermittelt³ und miteinander verglichen. Das Ergebnis ist eine Messreihe von Perioden-Kosten C_t pro Task, wobei der Index t die entsprechende Periode angibt. Die gemessenen Perioden-Kosten sind von Blockierungseffekten der Aktivitäten und von Scheduling-Entscheidungen nicht betroffen.

Abbildung 3.4: Messung der reinen Task-Kosten c_t

Gleitender Mittelwert

Eine simple Berechnung der mittleren Kosten (ACET) einer Task kann dann durch den *gleitenden Mittelwert* der Messreihe gemacht werden. Der gleitende Mittelwert (Durchschnitt) ist eine Methode zur Glättung von Zeit- bzw. Datenreihen, wobei aus einer ursprünglichen Zahlenmenge eine neue Zielmenge erstellt wird. Diese enthält die Mittelwerte von gleich großen Untermengen der ursprünglichen Datenreihe. Aus signaltheoretischer Sicht implementiert der gleitende

³Eine Möglichkeit der Messung reiner CPU-Kosten von Aktivitäten (Threads) innerhalb eines Linux-Kernels wird im Kapitel 4.4.2 vorgestellt.

Mittelwert einen Tiefpass-Filter⁴ mit endlicher Impulsantwort (FIR) [101] und wird definiert durch:

$$m_{MA}^{(n)}(t) = \frac{1}{n} \sum_{i=0}^{n-1} C_{t-i} \quad (3.5)$$

In der Gleichung 3.5 wird für jeden Wert t der Datenreihe der Mittelwert der letzten n Datenpunkten berechnet. Dabei definiert n die Größe eines gleitenden Fensters über die Zeitreihe, in dem die gemessenen Werte iterativ gemittelt werden. Somit bestimmt die Größe des Fensters die Stärke der Glättung, also das zugelassene Frequenzband des Tiefpass-Filters. Dieses Frequenzband kann zusätzlich mit Gewichtungen versehen werden, die die Prominenz einzelner Frequenzen im Mittelwert bestimmen [101]. Bei der Berechnung des gleitenden Mittelwerts müssen jedoch die letzten n gemessenen Werte zwischengespeichert werden. Die Komplexität des Algorithmus beträgt $\mathcal{O}(n)$, da hier bei jedem Schritt n Elemente verarbeitet werden müssen.

Exponentielle Glättung

Eine weniger rechen- und speicher-intensive Methode stellt die *exponentielle Glättung* dar, bei der Messwerte mit einer zunehmenden Aktualität eine höhere Gewichtung erhalten. Dabei werden den Datenpunkten einer Zeitreihe exponentiell abnehmende Gewichte zugeordnet. Durch eine ausgeglichene Alterung der Werte wird hier laufend ein Prognosewert ermittelt, wobei Schwankungen gleichmäßig auf die Schätzungen übertragen werden. Die exponentielle Glättung ist definiert durch die Gleichung 3.6.

$$y_t = \alpha \cdot C_t + (1 - \alpha) \cdot y_{t-1} \quad (3.6)$$

Im Zeitpunkt t wird für den aktuellen gemessenen Wert C_t ein geglätteter Schätzwert y_t errechnet, welcher sich als der gewichtete Durchschnitt aus dem aktuellen Wert C_t und dem letzten Schätzwert y_{t-1} ergibt. Der Schätzwert y_{t-1} wird auch als Historie bezeichnet. Die Gewichtung der Historie wird durch einen Glättungsfaktor α mit $0 \leq \alpha \leq 1$ bestimmt. Typischerweise wird für α ein relativ kleiner Wert gewählt (z.B. $\alpha = 0,125$), bei dem die Historie höher als der aktuell gemessene Wert gewichtet wird. Da diese Form der Glättung einen Filter mit unendlicher Impulsantwort (IIR) darstellt, hat sie gegenüber dem gleitenden Mittelwert eine wesentlich kürzere Verzögerung bei gleicher Glättung [101].

⁴Ein Filter zur Entfernung von hohen Frequenzanteilen aus einem Signal.

Ein entscheidender Vorteil der exponentiellen Glättung gegenüber dem gleitenden Mittelwert ist, dass in jedem Berechnungsschritt nur ein Wert für die Historie y_{t-1} zwischengespeichert werden muss. Darüber hinaus besitzt der Algorithmus eine geringere Komplexität von $\mathcal{O}(1)$, da der Aufwand für die Berechnung eines jeden Schritts konstant bleibt.

3.1.2 Schätzungen der Task-Kosten

Bildlich gesprochen schneidet eine Glättung die Spitzen der gemessenen Task-Kosten ab, also die Informationen über kurze und hochfrequente Schwankungen in den Task-Verbräuchen. Eine Regelung anhand von mittleren Werten birgt also die Gefahr von unvorhersehbaren transienten Engpässen in sich, die aufgrund des Fehlers in der Approximation der Tasks auftreten können. Für viele weiche Echtzeitsysteme vermag dieses Vorgehen ausreichend zu sein, z.B. wenn sie eine bestimmte Fristverletzungsrate tolerieren. Eine Regelung würde dann nur bei einer Verletzung dieser Rate eingreifen und die Qualität der Anwendungen optimistisch anpassen.

Wie bereits in Kapitel 1.4 diskutiert, soll die in dieser Arbeit entwickelte Ressourcensteuerung möglichst nah an die harte Echtzeit herankommen; d.h. Fristverletzungen sollen so weit wie möglich vermieden werden. Aus diesem Grund sind Schätzungen der Tasks notwendig, welche die Schwankungen (bzw. den *Jitter*) ihrer Verbräuche um den Durchschnitt ihrer Kosten sowie Tendenzen ihres Verhaltens berücksichtigen. Diese Schätzungen sollen dann zur Bestimmung von adaptiven Obergrenzen der realen Task-Kosten dienen. Auf diese Weise werden Engpässe zwar unwahrscheinlicher, die realen Kosten werden jedoch überprovisioniert. Ungenutzte Ressourcen sollen dann durch eine Regelung auf einer höheren Ebene des Modells kontrolliert an aktuelle Verbrauchs-Hotspots im System verteilt werden (siehe Kapitel 3.2).

Trend und Prognose des Task-Verhaltens

Anhand einer *doppelten exponentiellen Glättung* kann ein Trend und eine Prognose innerhalb der gemessenen Task-Kosten berechnet werden [102]. Die Prognose basiert hier auf ein Modell, welches aus einer Konstante und einem linearen Trend besteht. Die Gleichung 3.7 zeigt die Definition der Konstante a_t und des Trends b_t .

$$\begin{aligned} a_t &= \alpha \cdot C_t + (1 - \alpha) \cdot (a_{t-1} + b_{t-1}) \\ b_t &= \beta \cdot (a_t - a_{t-1}) + (1 - \beta) \cdot b_{t-1} \end{aligned} \tag{3.7}$$

Bei jedem Iterationsschritt werden hier nur zwei neue Werte berechnet, der Aufwand bleibt also konstant mit einer Komplexität von $\mathcal{O}(1)$. Darüberhinaus müssen nur zwei Werte für die Historien a_{t-1} und b_{t-1} gespeichert werden. Der Trend b_t definiert also einen Erwartungswert für die mittleren Task-Kosten innerhalb der nächsten Periode $t+1$. Dabei können die Prognose-Koeffizienten α und β unabhängig voneinander gesetzt werden. Ein Prognose-Wert p_n für den Zeitpunkt $t+n$, also nach n Perioden, kann anhand des linearen Modells durch die Gleichung 3.8 bestimmt werden:

$$p_n = y_t + b_t \cdot n \quad (3.8)$$

Mit dem linearen Term als Teil der Prognose verfolgt diese Methode Trends innerhalb der gemessenen Zeitreihe der Task-Kosten. Der Wert von p_n beinhaltet jedoch noch nicht den Jitter der Task-Kosten und kann deswegen nicht allein als Schätzung verwendet werden.

Erfassung des Jitters

Der Jitter der Task-Verbräuche etabliert sich aus den Schwankungen der gemessenen Kosten um ihren aktuellen Mittelwert. Der Jitter kann auf unterschiedliche Arten erfasst werden.

Ein naheliegender Ansatz, ist einen exponentiell geglätteten Wert für die Abweichung der gemessenen Kosten von ihrem aktuellen Mittelwert zu bestimmen:

$$J_t = \gamma \cdot |C_t - y_t| + (1 - \gamma) \cdot J_{t-1} \quad (3.9)$$

Hierbei ist C_t der aktuell gemessene Wert der Kosten einer Task zum Zeitpunkt t und y_t der exponentiell geglättete Mittelwert der Task-Kosten (nach der Gleichung 3.6). Diese Berechnung hat ebenfalls eine konstante Komplexität und benötigt wenig Speicherplatz.

Eine andere Methode zur Erfassung des Jitters ist die Berechnung der Standardabweichung über ein Fenster von n gemessenen Werten der Task-Kosten. Die Kosten einer Task werden dabei als eine Zufallsvariable $\mathcal{X}$ mit einer Normalverteilung repräsentiert (später gehen wir nochmal genauer auf die Verteilung ein). Die Standardabweichung definiert ein Maß für die Streuung der gemessenen Werten um ihren Erwartungswert (Mittelwert) und wird für die Task-Kosten zum Messzeitpunkt t definiert durch:

$$\sigma_n(t) = \sqrt{\frac{1}{n-1} \cdot \sum_{i=0}^{n-1} (C_{t-i})^2 - \frac{1}{n \cdot (n-1)} \cdot \sum_{i=0}^{n-1} (C_{t-i})^2} \quad (3.10)$$

Die Standardabweichung impliziert eine Gauß-Verteilung des Jitters der Task-Kosten. Das bedeutet, dass die realen Kosten der Tasks mit einer hohen Wahrscheinlichkeit in der unmittelbaren Umgebung ihres Mittelwerts erwartet werden. Wenn dies nicht der Fall ist, steigt der Wert der Abweichung abhängig von der Größe des Fensters n .

Kostenschätzung

Mit den bisherigen ermittelten Größen zur Approximation des Verhaltens einer Task kann nun eine Schätzung $C_{i,k,l}$ ihrer Kosten etabliert werden, welche einerseits automatisch Trends innerhalb der gemessenen Werten befolgt und auch Schwankungen um den Mittelwert der Kosten als Jitter auffasst.

Ermittelt man den Jitter der Task-Kosten wie in Gleichung 3.9, so kann nun mit einer Konstante Φ ein jitter-abhängiger Sicherheitspuffer über ihren exponentiell geglätteten Mittelwert y_t (vgl. Gleichung 3.6) definiert werden. Eine entsprechende Kostenschätzung für die k -te Task einer Anwendung A_i im Task-Modus l wird dann definiert durch:

$$C_{i,k,l} = \lceil y_t + \Phi \cdot J_t \rceil \quad (3.11)$$

Analog dazu kann eine Schätzung auch auf der Basis eines Prognose-Werts p_n (siehe Gleichung 3.8) des erwarteten durchschnittlichen Task-Verhaltens nach n Task-Perioden etabliert werden:

$$C_{i,k,l}^n = \lceil p_n + \Phi \cdot J_t \rceil \quad (3.12)$$

Für diese Variante der Schätzung ist es zunächst nicht klar mit welcher Güte der Jitter der Tasks abgedeckt wird, da dieser zunächst in Abhängigkeit von einem konstanten Glättungsfaktor γ (vgl. Gleichung 3.9) exponentiell geglättet wird. Für eine Anwendung mit seltenen, starken Ausreißern innerhalb der gemessenen Kosten kann ein größerer γ -Wert zur Bemessung des Jitter-Fensters von Vorteil sein, da er eine schnellere Anpassung des Mittelwerts an die realen Verbräuche ermöglicht. Bei Anwendungen deren Kosten konstant und gleich stark schwanken,

würde ein kleinerer γ -Wert die Historie höher gewichten und auch für einen stabileren Schätzwert sorgen.

Da J_t den Jitter nur durchschnittlich approximiert, wird mit Φ ein Sicherheitspuffer aufgebaut, der bestmöglich die realen Kosten einer Task abdecken soll (vgl. z.B. Gleichung 3.11). Dieser Faktor bestimmt, wie aggressiv eine Überprovisionierung der Kosten beim Auftreten von Schwankungen stattfinden soll, also wie reaktiv eine Schätzung auf Veränderungen der Kosten sein soll. Wird der Wert von Φ zu klein gewählt ist eine (sogar konstante) Unterprovisionierung der realen Kosten möglich, was zu unvorhersehbaren Engpässen führen kann. Ist Φ dagegen zu groß, erfolgt schon bei geringfügigen Schwankungen schnell eine Überprovisionierung.

Für eine möglichst genaue Approximation ist eine dynamische Anpassung von Φ notwendig. Diese kann durch einen Regelmechanismus (z.B. durch einen PID-Regler) realisiert werden, indem über Φ als Steuervariable ein minimaler Abstand zwischen den aktuell gemessenen Werten und der Schätzung eingestellt wird. Steigt der Jitter plötzlich an, so kann Φ aggressiv erhöht werden, um z.B. einen potentiellen Burst der Anwendung möglichst schnell zu erfassen. Fällt der Jitter wieder ab, z.B. wenn sich die Anwendung auf ein neues Verbrauchsniveau eingeepegelt hat, kann Φ wieder abgesenkt werden um die Überprovisionierung zu minimieren. Hierfür muss aber auch eine optimale Regelgeschwindigkeit eingestellt werden. In den meisten Fällen ist jedoch eine solche Regelgeschwindigkeit anwendungs- oder task-spezifisch und muss durch Experimente im Voraus bestimmt werden (vgl. Feedback Control Scheduling [45]).

Wählt man dagegen für die Bestimmung des Jitters die Methode über die Standardabweichung (vgl. Gleichung 3.10), so kann mit dem aktuellen Erwartungswert eine Schätzung erzeugt werden, die Schwankungen in den bereits gemessenen Task-Kosten mit einer bestimmten Güte abdeckt:

$$C_{i,k,l} = \lceil y_t + z \cdot \sigma \rceil \quad (3.13)$$

Für unterschiedliche Werte für z ergeben sich durch die entsprechenden Konfidenzintervalle für normalverteilte Zufallsvariablen folgende Werte für die Güte der Approximation:

z	% der Güte
1	68,3
2	95,4
3	99,7
4	99,9
...	...

Das bedeutet, dass ausgehend von der betrachteten Historie bei einer Kostenschätzung mit z.B. $C_{i,k,l} = \lceil y_t + 2 \cdot \sigma \rceil$ ca. 95,4 % der bereits gemessenen Task-Kosten unterhalb der Schätzung liegen. Die Kostenschätzung mit der Standardabweichung kann ebenfalls über einen Prognosewert (vgl. Gleichung 3.8) für den Mittelwert der nächsten n -Taskperioden gebildet werden:

$$C_{i,k,l}^n = \lceil p_n + z \cdot \sigma \rceil \quad (3.14)$$

3.1.3 Approximation des Verhaltens von Tasks

Eine einfache Regelung kann anhand des vorgestellten statistischen Modells aufgebaut werden, die die Schätzungen der Task-Kosten $C_{i,k,l}$ bei jeder gemessenen Periode, z.B. mit der Gleichung 3.14, aktualisiert. Aus Sicht der Ressourcenverteilung werden hiermit bei jeder neuen Task-Periode (statistisch) ausreichend Ressourcen reserviert, um eine erfolgreiche Ausführung der Tasklogik aus der beobachteten Vergangenheit mit einer bestimmten Güte zu garantieren.

Eine laufende Berechnung der Standardabweichung und der Erwartungswerte würde hier jedoch einen hohen Aufwand für ein reales System erzeugen. Ferner muss die Ressourcensteuerung die Planbarkeit der Tasks bei jeder Aktualisierung ihrer Schätzungen überprüfen und notfalls über ihre Qualität und die Verteilung der Ressourcen im System entscheiden. Welcher Aufwand dafür notwendig ist, wird später in Kapitel 3.1.4 genauer erläutert. Eine der Hauptvorgaben für ein ARTOS-System war jedoch, dass Rekonfigurationen der Anwendungsmodi so selten wie möglich stattfinden sollen (vgl. Kapitel 1.3). Aus diesem Grund wird hier ein Mechanismus vorgeschlagen, bei dem eine Reaktion der Steuerung nur dann ausgelöst wird, wenn eine aktuelle Annahme über das Verhalten einer Task nicht mehr zutreffend ist.

Abbildung 3.5: Konfidenzintervalle der Standardabweichung

Betrachtet man die Konfidenzintervalle der Standardabweichung in Abbildung 3.5, so sieht man dort die Anzahl der berücksichtigten Fälle in Prozent, deren Werte innerhalb eines entsprechenden Intervalls $\pm z \cdot \sigma_n$, mit $z = 1, 2, \dots, n$ liegen. Abgebildet auf das hier vorgestellte Kosten-Modell bedeutet das, dass so viel Prozent der gemessenen Task-Kosten in den vergangenen n -Perioden innerhalb dieses Approximationsbands lagen – das aktuelle Task-Verhalten wird also mit einer bestimmten Güte aus der unmittelbaren Vergangenheit von n Task-Perioden approximiert. Verwendet man zusätzlich Prognosewerte (vgl. Kapitel 3.1.2 „Trend und Prognose des Task-Verhaltens“) als Erwartungswerte für die Konfidenzintervalle, dann kann anhand eines ermittelten Trends innerhalb der Messreihe auch eine Approximation des Task-Verhaltens in der unmittelbaren Zukunft von n -Task-Perioden gemacht werden. Dabei wird der aktuelle Trend z.B. linear interpoliert, andere Vorgehensweisen sind jedoch auch möglich.

Ein Konfidenzintervall ergibt dann also einen *Sicherheitspuffer* (Safety Buffer (SB)) mit garantierten Ressourcen für eine Task

$$SB_{\pm}^+ = p_n \pm z \cdot \sigma_n \quad (3.15)$$

in dem zukünftige Schwankungen der tatsächlich gemessenen Kosten der Task mit einer bestimmten Güte toleriert werden. Eine konkrete Kostenschätzung wurde im letzten Abschnitt als die obere Schranke SB^+ dieses Sicherheitspuffers definiert. Die untere Schranke des Puffers SB_- verhindert eine konstante Überprovisionierung der Task-Kosten.

Wenn also die zukünftigen Kosten einer Task mit einer höheren Rate außerhalb des gewählten Sicherheitspuffers (also Konfidenzintervalls) fallen als erwartet, dann entspricht die aktuelle Approximation nicht mehr dem realen Anwendungsverhalten. In einem solchen Fall muss das Approximationsintervall aktualisiert werden, was auch ein neuer Wert für die Schätzung der Task-Kosten zufolge hat. Die Erfolgsrate, wie oft der aktuelle Sicherheitspuffer die realen Task-Kosten umgibt, kann in einem laufenden System sehr einfach als das arithmetische Mittel einer Abfolge von 1 (Kosten innerhalb Konfidenzintervall) und 0 (Kosten außerhalb Konfidenzintervall) bei jeder Task-Periode berechnet werden und mit der Vorgabe aus der Standardabweichung für die gewählte Puffergröße verglichen werden. Wird die Konfidenzrate nicht mehr erfüllt, so muss eine Reaktion und Überprüfung des Sicherheitspuffers der jeweiligen Task stattfinden.

Es gibt unterschiedliche Gründe warum die Approximation einer Task nicht mehr zutreffend sein kann. Diese werden im Folgenden kurz umrissen.

Anstieg bzw. Abfall der Kosten

Ein genereller Anstieg bzw. Abfall der Task-Kosten sollte sich bemerkbar machen durch eine Erhöhung der Rate, mit der die gemessenen Task-Kosten die Konfidenzgrenzen ihres aktuellen Sicherheitspuffers verletzen. Dabei ist es irrelevant, ob der Anstieg/Abfall stetig (*Step-Load*) oder abrupt (*Ramp-Load* bzw. *Burst*) passiert. Ein Burst äußert sich nur dahingehend, dass er schneller detektiert wird, da es hier schneller zu einer Verletzung der Trefferrate des Sicherheitspuffers kommt.

Ein Anstieg oder Abfall der Task-Kosten sollte sich generell auch in ihrem Erwartungs- und Prognosewert bemerkbar machen, wenn diese mit den Werten bei der letzten Aktualisierung des Sicherheitspuffers verglichen werden. Obwohl bei einer Aktualisierung anhand des aktuellen Prognosewerts die Anzahl der Task-Perioden potentiell interpoliert werden kann, bei der ein Sicherheitspuffer nicht mehr gültig ist, ist diese Art der Approximation nicht plausibel, weil eine Task ihr Verhalten innerhalb der Toleranzgrenzen verändern kann. Der Prognosewert im Aktualisierungszeitpunkt des Sicherheitspuffers ist also von relativ kurzfristigen Tendenzen der Task-Kosten betroffen, welche sich über die Zeit ausgleichen können, ohne dass eine Rekonfiguration des Puffers notwendig wird. In diesem Fall ist auch die Richtung einer potentiellen Verletzung des Sicherheitspuffers nur schwer vorauszusagen. Wenn es zu einer Verletzung der Konfidenzrate kommt, kann aufgrund der Veränderung des aktuellen Trends nur optimistisch bestimmt werden, ob es sich dabei um einen generellen Anstieg oder einen Abfall der Kosten handelt.

Wenn die Richtung der Veränderung jedoch von Belang ist, kann ein anderer Entscheidungsalgorithmus für die Aktualisierung eines SB implementiert werden, z.B. dann wenn die mittleren Task-Kosten y_t selbst den Sicherheitspuffer verletzen ($y_t < SB_-$ bzw. $SB^+ < y_t$). Ein Beispiel für eine solche Situation ist in Abbildung 3.9 (auf Seite 84) dargestellt. Bei dieser Variante ist es klar, dass das Verhalten einer Task stark genug von der Approximation abweicht, sodass ihre mittleren Kosten die Grenzen des Puffers in eine bestimmten Richtung verlassen. Allerdings wird dabei die Trefferquote des Sicherheitspuffers höchstwahrscheinlich schon früher verletzt, sodass die geforderte Approximationsgüte temporär nicht gewährleistet ist, bis die Änderung des Verhaltens detektiert wird. Sie birgt also theoretisch eine höhere Wahrscheinlichkeit für Engpässe. Die Konsequenz aus der Verletzung eines Sicherheitspuffers und die entsprechenden Vorgänge in ARTOS werden weiter im Kapitel 3.2.1 diskutiert.

Veränderung der stochastischen Verteilung der Kosten

Im vorgeschlagenen Modell wird zunächst angenommen, dass das Verhalten von Anwendungsaktivitäten normalverteilt ist. Dabei wird der Ressourcenverbrauch einer jeden Task pro Periode als eine stetige Zufallsvariable angenommen, für die alle Ereignisse, also die gemessenen Task-Ausführungszeiten, mit gleicher Wahrscheinlichkeit auftreten können. Durch das Modell werden also keine Kosten der Anwendungs-Tasks im Sinne ihrer Auftrittswahrscheinlichkeit bevorzugt. Hier wird von einer uniformen Verteilung der Belegungswahrscheinlichkeit des zugrunde liegenden stetigen Ressourcenraums ausgegangen.

Die Annahme der Normalverteilung beruht auf dem zentralen Grenzwertsatz welcher besagt, dass Zufallsvariablen, die eine additive Überlagerung vieler unabhängiger kleiner Einflüsse beschreiben, approximativ normal verteilt sind [88]. Dieses Modell ist zutreffend für das Verhalten von Tasks, deren Kosten von vielen einzelnen (unabhängigen) Faktoren (z.B. Caching-Effekte, Speicherzugriffszeiten) der aktuellen Ausführungsplattform beeinflusst sind. Darüber hinaus kann ja die Beschaffenheit der Berechnungsdaten unterschiedliche Komplexität hervorrufen. Ausgegangen von einer gleich-bleibenden Task-Logik würde sich also auf einer bestimmten Plattform für das jeweilige Szenario ein Erwartungswert⁵ etablieren, um welchen die gemessenen realen Task-Kosten pro Periode schwanken. Da diese Schwankungen eben von vielen zufälligen Faktoren abhängen, eignet sich in diesem Modell die Standardabweichung als Maß für die Streuung der Kosten um ihren Erwartungswert.

Nun kann es aber passieren, dass die Task-Logik dynamisch unterschiedliche Ausführungspfade einschlägt, z.B. aufgrund der Beschaffenheit von zu verarbeitenden Daten (z.B. auftretende I- und P-Frames in einem zu decodierenden Video-Strom, siehe Kapitel 2.3). Die Task-Kosten würden nun zwischen unterschiedlichen Häufungsbereichen mit unterschiedlichen Wahrscheinlichkeiten alternieren, die nicht mehr optimal durch eine Normalverteilung approximiert werden (sondern z.B. durch eine Binomialverteilung). Geht man nun davon aus, dass eine bisherige Approximation durch einen Sicherheitspuffer existiert, welcher durch die Standardabweichung zu einem früheren Zeitpunkt eingestellt wurde, dann kann eine Veränderung des Verhaltens und der Verteilung der Kosten zu einer höheren Verletzungsrate des Konfidenzintervalls führen als erlaubt. Dabei kann sich die Verhaltensänderung so etablieren, dass aus Sicht der Normalverteilung kaum eine Änderung der mittleren Kosten entsteht, die Verletzungsrate des Puffers jedoch trotzdem ansteigt.

⁵Da alle Werte die gleiche Wahrscheinlichkeit haben, reicht für den Erwartungswert das arithmetische Mittel der gemessenen Werte zu bestimmen

Wendet man nun trotzdem die Normalverteilung und die Standardabweichung für die Approximation des Task-Verhaltens an, so würde sich ein neuer Erwartungswert, mit einer relativ großen Streuung, zwischen den beiden Häufungspunkten etablieren. D.h. ein neuer Sicherheitspuffer würde um einen neuen Erwartungswert etabliert werden, der einen relativ großen Streuungsfaktor hat. Demzufolge wurde das neue Anwendungsverhalten mit der Standardabweichung und dem neuen Mittelwert auf der Basis der Normalverteilung kalibriert.

3.1.4 Problem der Planbarkeitsanalyse

Wie bereits in den vorherigen Kapiteln beschrieben, existiert innerhalb einer Plattform eine variable Menge von Applikationen mit internen Tasks, welche dynamisch in unterschiedlichen Modi aktiviert werden können. Darüber hinaus etablieren sich die tatsächlichen Ausführungszeiten der Anwendungsaktivitäten und somit die Ressourcenverbräuche der Anwendungen erst zur Laufzeit innerhalb einer bestimmten Zielplattform und diese können auch schwanken.

Die zentrale Zielvorgabe der hier entwickelten Ressourcensteuerung ist somit eine *aktive Konfiguration* der Anwendungsmodi auszuwählen und dabei:

1. den höchstmöglichen Nutzen bzw. Qualität im System zu erzeugen
2. unter Betrachtung der aktuellen Task-Verbräuche und der kumulativen Rechenzeitanforderung der Anwendungen
3. unter Betrachtung der verfügbaren Ressourcenmenge innerhalb einer Plattform
4. unter Betrachtung von Benutzerpräferenzen der Applikationen
5. und zum Schluss soll die Planbarkeit der Anwendungs-Tasks dynamisch geprüft und sichergestellt werden

Und speziell der letzte Punkt der Planbarkeitsanalyse stellt insofern ein größeres Problem dar, weil durch die Definitionsmenge von Task- und Anwendungsmodi die Komplexität für eine vollständige Planbarkeitsanalyse im System eskaliert. Ein herkömmlicher Ansatz wäre hier die Planbarkeit für alle Task-Modi Kombinationen zu überprüfen und dabei die beste Kombination nach den definierten Zielvorgaben auszuwählen, die auch einer gültigen Konfiguration von Applikationsmodi entspricht. Zur Erinnerung, ein Anwendungsmodus definiert sich als eine bestimmte vordefinierte Konfiguration von Betriebsmodi der einzelnen Anwendungs-Tasks (vgl. Kapitel 3.1). In diesem Fall wären dann also mindestens:

$$|\mathcal{M}_1| \times |\mathcal{M}_2| \times \dots \times |\mathcal{M}_n| \quad (3.16)$$

Planbarkeitstests für alle gültigen Modi-Konfigurationen der Anwendungs-Tasks notwendig, um die beste Konfiguration zu finden und dabei ihre temporale Korrektheit zu verifizieren. Selbst wenn hierfür ein sehr effizientes Verfahren zur Planbarkeitsanalyse verwendet wird, wie z.B. der *All Approximated*-Algorithmus von Albers et al. [29] (vgl. Kapitel 2.1.2), verbleibt hier immernoch die generelle Frage ungeklärt, *wann genau* und *wie oft* sollen diese Tests gemacht werden und ihre Komplexität in einem laufenden System in Kauf genommen werden?

Wie bereits in Kapitel 3.1.3 diskutiert wurde, wäre ein natürlicher Ansatz die Kostenschätzungen der Tasks bei jedem neu gemessenen Wert zu aktualisieren. Eine entsprechende Ressourcensteuerung würde dann die optimale Ressourcenverteilung im System bei jeder Änderung neu berechnen. In Kapitel 3.1.2 wurde jedoch bereits argumentiert, dass im vorgeschlagenen Modell die Aktualisierung einer jeden Schätzung mit einem Aufwand für die Berechnung der exponentiellen Glättung und der Standardabweichung der Task-Kosten verbunden ist. Dieser Aufwand wäre nun zusätzlich um die Komplexität für einen vollständigen Planbarkeitstest über alle Task-Modi-Kombinationen im System erweitert. In einem realen System ist ein solcher Aufwand jedoch nicht tragbar. Außerdem würde es mit einer solchen Regelung zu ständigen Umverteilungen der Ressourcen und somit zu frequentem Wechsel zwischen den verschiedenen Task- und Anwendungsmodi kommen (ähnlich wie beim *Imprecise Computations* [55] Ansatz, siehe Kapitel 2.2). Eine der zentralen Zielvorgaben für die entwickelte Ressourcensteuerung in Kapitel 1.3 war jedoch Rekonfigurationen der Modi so selten wie möglich zu betreiben, um zusätzliche Rekonfigurationskosten der Anwendungen zu minimieren.

Aus diesem Grund wurde im Rahmen dieser Arbeit auf der Basis der bereits diskutierten Task-Approximationen mit Sicherheitspuffern eine Strategie entwickelt, welche es ermöglicht, Planbarkeitstests in einem ARTOS-basierten System nur *dann, wenn* und *dort, wo* es nötig ist zu betreiben. Auf diese Weise wird die Komplexität für die Planbarkeitsanalyse in einem System zeitlich und räumlich verteilt. Dabei wurden anhand des vorgestellten Anwendungsmodells und der Approximation der Task-Verbräuche bereits Mechanismen etabliert, welche die Basis für eine Reaktion auf eine Verhaltensänderung einer Applikation schaffen (vgl. Kapitel 3.1.3). In einem solchen Fall muss die Vorhersage des Verhaltens aktualisiert werden, wobei die Planbarkeit zunächst nur stationär für die betroffene Anwendung überprüft werden muss. Dies kann jedoch zu einer Optimierung der Ressourcenverteilung im Gesamtsystem führen, was wiederum auf einer *höheren Ebene* entschieden werden muss. Diese Vorgehensweise wird durch das *hierarchische*

Scheduling-Modell von ARTOS vorgegeben, welches nun im folgenden Kapitel 3.2 vorgestellt und dessen Funktionsweise genau erläutert wird.

3.2 Hierarchisches Scheduling-Modell

Der vorgeschlagene Ansatz zur Reduktion des Aufwands bei der Approximation des Task- und Anwendungsverhaltens, sowie zur zeitlichen und räumlichen Verteilung der Komplexität der zur Laufzeit notwendigen Planbarkeitsanalyse in einem System, wird durch ein spezielles hierarchisches Scheduling-Modell unterstützt. Dieses Modell befolgt die Abstraktionen von Tasks, Applikationen und unterschiedlichen Betriebsmodi auf verschiedenen Ebenen, wie in Abbildung 3.6 dargestellt. Dabei wird das Anwendungsverhalten auf der untersten Ebene überwacht und ähnlich wie bei einer *kaskadierten Regelung* [103] iterativ an die höheren Ebenen weitergereicht, wobei es innerhalb jeder Ebene abstrahiert und die übermittelte Datenmenge reduziert wird. Somit werden dann auf der höchsten Ebene nur noch globale Entscheidungen über die Konfiguration der Anwendungsmodi getroffen.

Abbildung 3.6: Ebenen des hierarchischen Scheduling-Modells von ARTOS

Abbildung 3.7 veranschaulicht das grobe Vorgehen innerhalb dieses Modells für ein darauf basierendes System zur Laufzeit. Auf der untersten Ebene, der sog. *Task-Ebene*, sind applikations-lokale *Task-Scheduler* angesiedelt – einer pro Applikation. Diese steuern die Ausführung der Aktivitäten innerhalb der Anwendungen mit EDF und approximieren ihr Verhalten anhand ihrer periodisch gemessenen Ausführungskosten (vgl. Abbildung 3.7). Aus diesen Approximationen werden zunächst Schätzungen für die Task-Kosten ermittelt, welche auch Schwankungen im

Task-Verhalten abbilden. Anhand dieser Schätzungen werden die Ressourcenanforderungen der Anwendungen in ihren aktuellen Betriebsmodi abgeleitet. Die benötigten Ressourcenkapazitäten werden dann auf der *Applikations-Ebene* reserviert und auch durchgesetzt. Dabei sorgt das Modell auf der *System-Ebene* im Zusammenspiel mit den unteren Ebenen automatisch dafür, dass eine Unter- bzw. Überprovisionierung der verfügbaren Systemressourcen möglichst nicht stattfindet. Es wählt stets eine optimale Konfiguration der Anwendungsmodi unter Betrachtung ihrer (aktuell) ermittelten Ressourcenverbräuchen und der Anwendungsprioritäten aus. Für eine Minimierung des Scheduling-Aufwands ist es nun entscheidend, wann Informationen zwischen den Ebenen ausgetauscht und Aktionen angestoßen werden.

Abbildung 3.7: Approximierter stabiler Zustand

Ausgegangen von einer zur Laufzeit bereits eingestellten und zunächst optimalen Konfiguration der Anwendungsmodi, solange die Approximationen auf der Task-Ebene akkurat bleiben, wird keine Zustandsänderung zur nächst-höheren Ebene propagiert. In diesem Fall bleibt das System planbar und der Overhead für die Berechnung bzw. Aktualisierung der Kostenschätzungen der Tasks und für die Planbarkeitsanalyse innerhalb des Systems wird gespart. Als einziger Aufwand verbleibt lediglich das Prüfen der gemessenen Task-Kosten gegen ihre Approximationen auf der Task-Ebene. Nur wenn eine Task ihre Approximationsgrenzen verletzt⁶ (vgl. Abbildung 3.8), wird ihre Kostenschätzung ($C_{i,k,l}$) aktualisiert und an die Applikations-Ebene zugestellt. Dabei wird eine lokale Planbarkeitsanalyse für die betroffene Anwendung durchgeführt und ihre Ressourcenanforderung ($R_{i,j}$) im aktiven Betriebsmodus aktualisiert. Da hierbei eine Unter-

⁶dabei durchstoßen die mittleren Kosten einer Task die obere oder die untere Grenze ihres aktuellen Sicherheitspuffers (siehe Kapitel 3.1.3)

bzw. Überprovisionierung der global verfügbaren Systemressourcen entstehen kann, wird auf der *System-Ebene* die Ressourcenbelegung durch die aktuelle Konfiguration der Anwendungsmodi überprüft und unter Umständen optimiert. Die Aktualisierung der Kostenschätzung einer Task kann also zu einer neuen Ressourcenverteilung der Anwendungen und somit Rekonfiguration des Gesamtsystems führen. Dabei kann einerseits eine Degradierung der Qualität bestimmter Anwendungen stattfinden, wenn zum Beispiel ein Engpass detektiert wurde, oder aber auch eine Verbesserung der Gesamtqualität des Systems, falls durch die Aktualisierung genügend Ressourcen für höhere Anwendungsmodi frei geworden sind. Die Vorgaben für die Auswahl der optimalen Konfiguration der Betriebsmodi der Anwendungen wurden bereits in Kapitel 3.1.4 grob vorgestellt.

In den folgenden Kapiteln wird nun ein tieferer Einblick in die einzelnen Ebenen des Scheduling-Modells von ARTOS gewährt.

Abbildung 3.8: Zustandsaktualisierung

3.2.1 Scheduling auf der Task-Ebene

Wie bereits erklärt besitzt jede Anwendung auf der Task-Ebene einen Task-Scheduler, welcher die Aktivierung ihrer internen Tasks innerhalb einer bestimmten Ressourcenkapazität ($R_{i,j}$) steuert und ihre aktuellen Kosten approximiert. Die Ressourcenkapazitäten für die Anwendungen werden im vorgestellten Modell von der Applikations-Ebene abhängig von ihren aktuellen Betriebsmodi vorgegeben. Dies geschieht anhand einer periodischen Reservierung und dynamischer Kalibrierung einer bestimmten Menge an Rechenzeit für jede Anwendung. Das Schedu-

ling auf der Anwendungsebene ist darauf ausgelegt, die reservierten Anwendungskapazitäten durchzusetzen. Die Planbarkeit der Task-Menge einer jeden Anwendung muss also anhand der aktuellen Kostenschätzungen der Tasks innerhalb der ihr zugewiesenen Rechenzeitkapazität sichergestellt werden. Die Rechenzeitkapazität muss jedoch auch aktualisiert werden, wenn sich Kostenschätzungen der Tasks ändern. Der genaue Mechanismus wird später im Kapitel 3.2.2 näher erläutert.

Scheduling-Mechanismus

Aufgrund der unvorhersehbaren und schwankenden Ausführungszeiten der Anwendungsaktivitäten ergibt sich abhängig vom verwendeten Scheduling-Mechanismus auf der Task-Ebene dynamisch ein konkreter Aktivierungsablauf der Anwendungs-Tasks. Dieser kann nicht im Voraus (offline) eingeplant und somit auf Validität, im Sinne der Erfüllung der Echtzeitanforderungen der Anwendung, getestet werden. Die Planbarkeit einer jeden Anwendung muss also auf der Basis von Kostenschätzungen der Tasks über die Auslastung der ihr zugewiesenen Ressourcen-Kapazität periodisch zur Laufzeit verifiziert werden.

Prinzipiell kann innerhalb eines Task-Schedulers eine beliebige, applikationsspezifische Aktivierungsstrategie verwendet werden. Diese Strategie ist in erster Linie zuständig für die Auswahl der nächsten auszuführenden Task vor allem dann, wenn zu einem bestimmten Zeitpunkt mehrere Tasks gleichzeitig zur Aktivierung bereit stehen, d.h. also um Rechenzeit innerhalb ihrer Anwendung konkurrieren. In der Echtzeittheorie existieren mehrere Rechenzeit-Zuteilungsalgorithmen für Uniprozessoren (vgl. [22]), die hier verwendet werden können, wie z.B. Rate Monotonic Scheduling (RMS), Earliest Deadline First (EDF), Least Laxity First (LLF), usw. Diese unterscheiden sich zunächst in ihrer Optimalität und im Verwaltungsaufwand zur Laufzeit. Sie implizieren jedoch auch eine unterschiedliche Komplexität für den Planbarkeitsnachweis einer Task-Menge.

Im Rahmen dieser Arbeit wurde für das Scheduling auf der Task-Ebene ein Earliest Deadline First-Mechanismus gewählt. Wie in Kapitel 2.1.1 bereits diskutiert wurde, ist EDF ein optimales Verfahren, welches eine gültige Aussage über die Planbarkeit einer Taskmenge auch bei einer hohen Auslastung der verfügbaren Ressourcenmenge ermöglicht [22]. Darüberhinaus besitzt EDF die Eigenschaft, dass sich die Aktivierungsreihenfolge innerhalb der Hyperperiode der Tasks wiederholt. Diese Eigenschaft wird später für die anwendungsspezifische Planbarkeitsanalyse und zugleich Kapazitätsermittlung auf der Applikations-Ebene ausgenutzt (siehe Kapitel 3.2.2).

Sicherheitspuffer

Aus der Sicht eines Task-Schedulers sind die Ausführungszeiten der Anwendungsaktivitäten zunächst irrelevant, da hier stets die Task mit der kürzesten Frist als nächste drankommt. Die tatsächlichen Ausführungszeiten sind nur bei der Planbarkeitsanalyse von Belang, also bei der Überprüfung ob alle Tasks einer Anwendung mit ihren aktuell ermittelten Kosten innerhalb der gegenwärtigen Rechenzeitkapazität der Anwendung ihre Fristen einhalten werden. Hierbei werden Kostenschätzungen der Tasks auf der Basis von Approximationen verwendet, die anhand eines jitter-abhängigen Sicherheitspuffers für ihre Schwankungen etabliert werden (vgl. Kapitel 3.1.2, Gleichung 3.14). Diese Schätzungen sind gegenüber den tatsächlich gemessenen Werten der Task-Ausführungszeiten relativ stabil, weil sie sich nur bei starken Schwankungen und Abweichungen vom erwarteten Task-Verhalten verändern. Es ist naheliegend, dass eine Planbarkeitsanalyse für eine Anwendung nur dann erneut durchgeführt werden muss, wenn sich die Kostenschätzungen ihrer internen Tasks ändern.

In Kapitel 3.1.3 wurde bereits erklärt, dass mit der Standardabweichung der gemessenen Kosten einer Task über eine Zeitspanne von n -Perioden ein Konfidenzintervall um ihren aktuellen Mittelwert aufgespannt werden kann. Dieses Konfidenzintervall definiert einen sog. Sicherheitspuffer (SB), welcher, wie in Abbildung 3.9, das Verhalten der Task aus der unmittelbaren Vergangenheit mit einer bestimmten Güte approximiert. Mit der doppelten Standardabweichung $2 \cdot \sigma$ werden zum Beispiel 95,4% der gemessenen Task-Kosten durch das Konfidenzintervall abgedeckt. Der entsprechende Sicherheitspuffer wird dann anhand des aktuellen Erwartungswerts der Task-Kosten y_t konstruiert:

$$\begin{aligned} C_{i,k,l} &= SB_{max}^n = \lceil y_t + 2 \cdot \sigma \rceil \\ SB_{min}^n &= \lfloor y_t - 2 \cdot \sigma \rfloor \end{aligned} \tag{3.17}$$

wobei n die Größe des Fensters in Task-Perioden angibt, innerhalb welchem die Standardabweichung der gemessenen Task-Kosten bestimmt wird. Die aktuelle Kostenschätzung $C_{i,k,l}$ der Task entspricht dann der oberen Grenze SB_{max} des Sicherheitspuffers, die mit der vorgegebenen Güte Schwankungen im aktuellen Ressourcenverbrauch der Task mit Ressourcen versorgen kann. Die untere Grenze des Puffers SB_{min}^n verhindert eine konstante Überprovisionierung der Ressourcen, zum Beispiel, wenn der Ressourcenverbrauch der Task permanent abfällt.

Abbildung 3.9: Sicherheitspuffer mit $2 \cdot \sigma$

Aktualisierung der Sicherheitspuffer

Der Sicherheitspuffer dient außerdem zur Feststellung von Abweichungen von dem erwarteten Task-Verhalten, z.B. dann wenn die gemessenen Task-Kosten mit einer bestimmten Rate außerhalb des Puffers liegen (vgl. Abbildung 3.9). Bei einem zum Beispiel mit der doppelten Standardabweichung etablierten Konfidenzintervall müssen demnach mindestens 4.6% der gemessenen Task-Kosten außerhalb des aktuellen Puffers liegen, damit die Güte-Bedingung der Approximation verletzt wird. In diesem Fall wird eine Aktion des Schedulers ausgelöst und die Approximation aktualisiert.

Ein möglicher Zeitpunkt zur Feststellung einer Abweichung vom approximierten Zustand ist, wenn der Wert der mittleren Task-Kosten y_t die Grenzen des Sicherheitspuffers durchstößt (siehe Abbildung 3.9). In diesem Fall kann davon ausgegangen werden, dass die geforderte Approximationsgüte, also die Trefferrate der gemessenen Task-Kosten innerhalb des aktuellen Konfidenzintervalls, mit Sicherheit bereits verletzt wurde. Mit dieser Variante kann jedoch auch eine Aussage über einen generellen Anstieg oder Abfall der Kosten gemacht werden, je nachdem ob die obere oder untere Grenze des Puffers überschritten wurde. Auf einer höheren Ebene kann diese Information wiederum verwendet werden, um zwischen einem Engpass und einer Ressourcenfreigabe

zu unterscheiden.

Die Approximation der Task-Kosten mit dem Konfidenzintervall entspricht also einer Momentaufnahme des Task-Verhaltens zum Zeitpunkt der letzten Puffer-Aktualisierung. Bei einer Abweichung von dieser, muss der Sicherheitspuffer und die Kostenschätzung der Task, wie in Abbildung 3.9, aktualisiert werden. Dabei wird zunächst:

1. die Standardabweichung σ der gemessenen Task-Kosten innerhalb der letzten n -Perioden neu berechnet
2. ein neuer Sicherheitspuffer mit $SB_{max} = y + 2 \cdot \sigma$ und $SB_{min} = y - 2 \cdot \sigma$ anhand des aktuellen Erwartungswerts y_t und der errechneten Standardabweichung σ etabliert
3. eine neue Kostenschätzung $C_{i,k,l} = SB_{max}$ gespeichert und eine Aktivität auf der Applikations-Ebene ausgelöst

Wurde eine Aktualisierung der Kostenschätzung einer Task $\tau_{i,k}$ vorgenommen, dann ist die Planbarkeit der gesamten Taskmenge $\mathcal{T}_i$ einer Anwendung A_i mit ihrer aktuellen Ressourcenkapazität $R_{i,j}$ nicht gewährleistet und muss überprüft werden. Unter Umständen wird aber auch die von den Anwendungs-Tasks benötigte Ressourcenmenge auf der Applikationsebene gerade überprovisioniert. Die Ressourcenreservierung für die gesamte Anwendung muss nun neu eingestellt werden. Dies kann wiederum, wie bereits erklärt, eine Optimierung der Gesamtkonfiguration der Anwendungsmodi im System auslösen, z.B. wenn ein Engpass entsteht oder Ressourcen für potentiell höhere Modi frei geworden sind.

Die Verletzung eines Sicherheitspuffers ist also mit einem Berechnungsaufwand für die Standardabweichung, für die neue Kostenschätzung und den neuen Puffer, für die Planbarkeitsanalyse auf der Anwendungsebene und für die eventuelle Optimierung und Umschaltung der aktuellen Modi-Konfiguration der Anwendungen verbunden. Im Kapitel 4.4.7 wird im Rahmen der Implementierung des Modells eine Möglichkeit vorgestellt, wie dieser Aufwand innerhalb eines ARTOS-Systems zur Laufzeit dynamisch erfasst und ebenfalls eingeplant werden kann.

Instantane Aktualisierung der Sicherheitspuffer

In Abbildung 3.10a ist eine *instantane Aktualisierung* eines Sicherheitspuffers dargestellt. Diese findet sofort statt, wenn die mittleren Task-Kosten y eine der Grenzen des aktuellen Puffers ($SB_{min,max}$) durchstoßen. Eine Überprüfung, ob dies der Fall ist, findet bei jeder Aktualisierung

der mittleren Kosten statt – also bei jeder Task-Periode. Diese Vorgehensweise birgt verhältnismäßig einen höheren Berechnungsaufwand und eine hohe Rekonfigurationswahrscheinlichkeit für ein System, da hiermit Sicherheitspuffer tendenziell öfters aktualisiert werden. Adererseits wird damit das Task-Verhalten genauer approximiert.

Abbildung 3.10: Aktualisierung eines Sicherheitspuffers (aus [15])

Periodische Überprüfung der Sicherheitspuffer

Durch eine *periodische Überprüfung* der Sicherheitspuffer (siehe Abbildung 3.10b) kann der Aktualisierungsaufwand auf Kosten der Approximationsgenauigkeit reduziert werden. Bei diesem Vorgehen wird die Einhaltung der Puffergrenzen durch die mittleren Kosten periodisch mit einem Abstand von n Task-Perioden überprüft. Eine plausible Strategie dafür erscheint die Überprüfung der Sicherheitspuffer bei jeder Hyperperiode der Tasks innerhalb einer Anwendung, da durch die Kapazitätsbemessung auf der Applikations-Ebene ihre Planbarkeit zunächst innerhalb dieser sichergestellt wird (vgl. Kapitel 3.2.2). Auf diese Weise kommt es zwar seltener zu Rekonfigurationen im System, es können jedoch wahrscheinlicher Fristverletzungen auftreten, da das Task-Verhalten zeitweise nicht korrekt approximiert sein kann – z.B. dann, wenn die mittleren Kosten aus den Puffergrenzen hinausragen und eine Aktualisierung des Puffers noch nicht stattgefunden hat. In einem solchen Fall kann es passieren, dass eine vorzeitige Erschöpfung der reservierten Anwendungsressourcen vor dem Auftreten der nächsten Hyperperiode stattfindet und nicht alle Tasks mit genügend Rechenzeit bedient werden können.

Dynamische Strategie für die Puffer-Aktualisierung

Um einen Trade-off zwischen beiden Strategien für die Pufferaktualisierung in ARTOS zu treffen wurde folgende Vorgehensweise gewählt:

- **Tasks mit Fristverletzungen:**

Wenn Tasks unter Fristverletzungen leiden, wird dynamisch auf die *instantane* Pufferaktualisierung umgeschaltet, da hier das Task-Verhalten besser approximiert und die Fristverletzungsrate abgesenkt werden soll. Dabei wird auch die anwendungsspezifische Ressourcenkapazität häufiger den Verbräuchen der Tasks entsprechend aktualisiert.

- **Tasks ohne Fristverletzungen:**

Leiden Tasks nicht unter Fristverletzungen, dann wird auf die *periodische* Pufferaktualisierung umgeschaltet; hier scheint die Approximation der Tasks innerhalb der Anwendung noch gut genug zu sein, also soll der Aufwand im System reduziert werden.

Die Implementierung beider Strategien zur Überprüfung und Aktualisierung der Sicherheitspuffer innerhalb eines ARTOS-Frameworks wird im Kapitel 4.4.7 weiter ausgeführt. Im folgenden Kapitel wird nun auf das Scheduling auf der Applikations-Ebene und speziell auf die Bemessung der benötigten Ressourcenkapazität für jede Anwendung eingegangen.

3.2.2 Scheduling auf der Applikations-Ebene

Abbildung 3.11: Kapselung von Anwendungen in Constant Bandwidth Servern (aus [15])

Auf der Applikations-Ebene wird jede Anwendung durch einen *Constant Bandwidth Server* [31] (CBS) gekapselt und damit ihr Ressourcenverbrauch isoliert (vgl. Abbildung 3.11). Die CBS sind eine budget-orientierte Erweiterung des Earliest Deadline First Scheduling-Mechanismus. Sie dienen im vorgestellten Modell als Abstraktionsschicht zwischen den geschätzten Kosten auf der Task-Ebene und der Entscheidung über die aktuellen Betriebsmodi der Anwendungen auf der System-Ebene.

Die Hauptaufgabe eines Constant Bandwidth Servers ist die *temporale Isolation* von Tasks. Dabei soll das Einhalten von Fristen innerhalb einer Task nicht durch die Ausführung anderer Tasks beeinflusst werden können. In einem klassischen CBS-basierten System wird also jeder Task ein eigener CBS zugewiesen, welcher ihre Ausführung steuert und eine Ressourcenkapazität periodisch durchsetzt. Der CBS verhält sich dabei so, als würde die Task von einer separaten *virtuellen CPU* mit niedrigerer Taktfrequenz ausgeführt werden.

Ein Constant Bandwidth Server wird charakterisiert durch eine Periode (und gleichzeitig Deadline) und eine Kapazität (Budget) an Ausführungszeit. Das Verhältnis zwischen Kapazität und Periode ergibt eine *Ressourcenbandbreite*, also den Anteil an CPU-Zeit, welcher für eine gekapselte Task innerhalb jeder Server-Periode reserviert werden soll. Besitzt die bediente Task eine eigene Periode und Deadline, so obliegt es der Konfiguration ihres Servers genügend Bandbreite zu reservieren, damit sie ihre Fristen einhält. Steigt der Ressourcenverbrauch der Task an, so leidet durch die Begrenzung mit dem CBS zunächst nur sie selbst unter einer Unterprovisionierung und potentiellen Fristverletzungen.

Der klassische CBS-Mechanismus ist zunächst für die Kapselung einer einzigen Task innerhalb eines Servers ausgelegt. Wenn in einem System der Ressourcenverbrauch aller Anwendungs-Tasks kontrolliert und voneinander isoliert werden soll, müssen diese in einzelnen CBS isoliert werden. Dies bedeutet wiederum einen erhöhten Scheduling-Aufwand zur Laufzeit. Soll zusätzlich noch der Ressourcenverbrauch ganzer Anwendungen voneinander getrennt und nach Benutzerprioritäten gesteuert werden, müssen die einzelnen Task-Server auf einer höheren Abstraktionsebene konfiguriert und überwacht werden, was einen zusätzlichen Scheduling-Aufwand zur Laufzeit erzeugt.

In dieser Arbeit wurde darum auf der Basis der CBS ein Mechanismus entwickelt, welcher die Kapselung einer ganzen Anwendung mit all ihren Tasks innerhalb eines einzigen Servers möglich macht. Der Server wird dann dynamisch so dimensioniert, dass er alle Tasks der gekapselten Anwendung mit der gleichen Ressourcenbandbreite bedienen und somit den Verbrauch der gesamten Anwendung isolieren kann. Dies bedarf einiger Anpassungen des ursprünglichen CBS Mechanismus und seiner Grundannahmen, welche im Laufe dieses Kapitels vorgestellt werden.

Die exakte Funktionsweise der CBS ist also für das Verständnis der Vorgänge auf der Applikations-Ebene eines ARTOS-Schedulers maßgeblich und wird darum im Folgenden genau erklärt.

Funktionsweise eines Constant Bandwidth Servers

Der Constant Bandwidth Server ist ein Scheduling-Mechanismus, mit dem Auswirkungen von Kostenüberläufen (*Overruns*) auf ihre Verursacher eingeschränkt werden können. Er findet generell dann Anwendung, wenn die genauen Kosten von nicht-zeitkritischen (auch aperiodischen) Aktivitäten nicht bekannt sind, um sie neben zeitkritischen (periodischen) Tasks innerhalb eines Echtzeitsystems einzuplanen. In dieser Arbeit wird der CBS-Mechanismus später erweitert, um mehrere zeit-kritische (periodische) Tasks gleichzeitig mit einem Server bedienen zu können.

Nach [31] ist ein Constant Bandwidth Server zunächst wie folgt definiert (die Definition wurde bereits in einer ARTOS-spezifischen Notation übertragen):

1. Ein CBS wird charakterisiert durch ein Budget c_s und eine Bandbreite $B_{i,j} = R_{i,j}/P_{i,j}$, also durch eine maximale Rechenzeitkapazität $R_{i,j}$ pro Serverperiode $P_{i,j}$. Zu jedem Zeitpunkt besitzt der Server eine absolute Frist $D_{i,j}$. Zu Beginn ist die Server-Frist $D_{i,j} = 0$.
2. Jedem ausgeführten Task-Job $J_{i,k,l}$ wird dynamisch eine Frist $d_{i,k,l}$ zugewiesen, die der aktuellen Server-Frist $D_{i,j}$ entspricht ($d_{i,k,l} = D_{i,j}$).
3. Wenn ein Job ausgeführt wird, wird das Budget c_s um die gleiche Rechenzeit dekrementiert.
4. Wenn $c_s = 0$, dann wird das Budget sofort wieder auf den maximalen Wert $R_{i,j}$ aufgeladen und eine neue Server-Frist $D_{i,j} = D_{i,j} + P_{i,j}$ generiert. Somit existieren keine Intervalle in denen der Server ein Budget gleich Null hat.
5. Ein Server ist zu einem Zeitpunkt t *aktiv*, wenn ausstehende Task-Jobs existieren; dies ist ebenfalls der Fall, wenn gerade ein Job $J_{i,k,l}$ ausgeführt wird. Der Server ist *inaktiv*, wenn er nicht aktiv ist.
6. Wenn ein neuer Task-Job $J_{i,k,l}$ eintrifft und der Server ist bereits aktiv, dann wird dieser in eine Warteschlange entsprechend einer nicht-präemptiven Disziplin eingereiht (z.B. FIFO).
7. Wenn zum Zeitpunkt $r_{i,k,l}$ ein neuer Job $J_{i,k,l}$ eintrifft und der Server ist inaktiv, wenn $c_s \geq (D_{i,j} - r_{i,k,l}) \cdot B_{i,j}$ dann wird eine neue Deadline $D_{i,j} = r_{i,k,l} + P_{i,j}$ generiert und c_s wird wieder auf den maximalen Wert $R_{i,j}$ aufgeladen. Anderenfalls wird der Job mit der letzten Server-Deadline $D_{i,j}$ und dem aktuellen Budget c_s bedient.

8. Wenn ein Job beendet wird, dann wird der nächste ausstehende Job mit dem aktuellen Budget c_s und der aktuellen Deadline $D_{i,j}$ des Servers bedient. Wenn kein solcher Job vorhanden ist, bleibt der Server inaktiv.
9. Zu jedem Zeitpunkt wird einem Job die letzte generierte Server-Deadline zugewiesen.

Die CBS werden innerhalb eines Systems anhand ihrer aktuellen Deadlines $D_{i,j}$ mit EDF aktiviert. Die oben aufgeführten Regeln für das CBS-Scheduling sorgen automatisch für die Einhaltung der reservierten Server-Bandbreiten und für temporale Isolation. Die Verschiebung einer Server-Deadline nach vorn auf der Zeitachse (siehe Regel 4) gleicht einer Prioritätsverminderung eines aktiven Servers. Aufgrund von EDF kann nun ein anderer Server mit einer kürzeren Frist als nächstes dran kommen, der ursprünglich aktive CBS kann also potentiell verdrängt werden. Die Planbarkeit eines CBS-basierten Systems wird durch einen einfachen Auslastungstest, mit einer konstanten Berechnungskomplexität, über die Summe der Server-Bandbreiten überprüft:

$$\sum_{i=1}^n B_{i,j_i} \leq 1 \quad (3.18)$$

Die Überprüfung, ob beim Eintreffen eines Jobs $J_{i,k,l}$ innerhalb eines inaktiven Servers $c_s \geq (D_{i,j} - r_{i,k,l}) \cdot B_{i,j}$ ist (siehe Regel 7), wird als *Aufweckregel* (Wake-up Rule) bezeichnet [104]. Dabei überprüft der Scheduler, ob der benötigte Rechenzeitanteil zur Ausführung des eintreffenden Jobs (gegeben durch das aktuelle Rechenzeit-Budget c_s geteilt durch das Zeitintervall $D_{i,j} - r_{i,k,l}$ von jetzt bis zur aktuellen Deadline) größer ist als die Server-Bandbreite $R_{i,j}/P_{i,j}$:

$$c_s \geq (D_{i,j} - r_{i,k,l}) \cdot B_{i,j} \Rightarrow \frac{c_s}{D_{i,j} - r_{i,k,l}} \geq \frac{R_{i,j}}{P_{i,j}} \quad (3.19)$$

Wenn dies der Fall ist, dann kann der aktuelle Task-Job nicht mit dem verbleibenden Budget und der aktuellen Server-Deadline bedient werden, weil er dann mehr Rechenzeit als $B_{i,j} = R_{i,j}/P_{i,j}$ verbrauchen und die temporale Isolation seines Servers verletzen würde. Als Lösung wird dann eine neue Scheduling-Deadline $D_{i,j} = r_{i,k,l} + P_{i,j}$ generiert und das Budget auf $c_s = R_{i,j}$ aufgeladen, womit die temporale Korrektheit des Schedulers gewährleistet bleibt [104].

Die Funktionsweise eines Constant Bandwidth Servers wird nun anhand eines aus [31] entnommenen Beispiels veranschaulicht (die Notation wurde hier bewusst vereinfacht). Eine zeitkritische Task τ_1 wird zusammen mit einer Task τ_2 ausgeführt, wobei τ_2 von einem CBS mit einem Budget $R = 2$ und einer Periode von $P = 7$ Zeiteinheiten, also mit einer Bandbreite von $B = \frac{2}{7}$, bedient wird. Die Aktivierungsreihenfolge beider Tasks ist in Abbildung 3.12 exemplarisch dargestellt.

Abbildung 3.12: Funktionsweise eines CBS (Beispiel entnommen aus [31])

Der erste Job von τ_2 trifft zum Zeitpunkt $r_1 = 2$ ein während der Server inaktiv ist. Mit $c_s \geq (D_0 - r_1) \cdot B$ bekommt der Server (und somit auch die Task) eine Deadline $D_1 = r_1 + P = 9$ zugewiesen und seine Kapazität c_s wird auf $R = 2$ aufgeladen. Zum Zeitpunkt $t = 6$ ist die Server-Kapazität aufgebraucht, also wird eine neue Server-Deadline $D_2 = D_1 + P = 16$ generiert und c_s wird wieder voll aufgeladen. Zum Zeitpunkt r_2 trifft ein zweiter Job von τ_2 ein während der Server bereits aktiv ist, also wird dieser in die Warteschlange eingefügt. Wenn der erste Job bei $t = 9$ terminiert, wird der zweite Job mit der aktuellen Server-Deadline ($D_2 = 16$) und der verbleibenden Kapazität ($c_s = 1$) bedient. Zum Zeitpunkt $t_2 = 12$ ist die Server-Kapazität erneut aufgebraucht, also wird wieder eine neue Server-Deadline $D_3 = 23$ generiert und das Budget auf $R = 2$ aufgefüllt. Der dritte Job von τ_2 trifft zum Zeitpunkt $r_3 = 17$ ein, wenn der Server bereits inaktiv ist. Mit $c_s = 1 < (D_3 - r_3) \cdot B = (23 - 17) \cdot \frac{2}{7} = 1.71$ wird der Job nun mit der aktuellen Server-Deadline D_3 und dem aktuellen Budget $c_s = 1$ bedient.

Anforderungen für die Kapselung mehrerer Tasks innerhalb eines CBS

Betrachtet man das vorherige Beispiel, so kann grundsätzlich angenommen werden, dass statt mehrere Jobs ein und derselben Task, nun Jobs unterschiedlicher Tasks einer Anwendung innerhalb eines CBS eintreffen und ausgeführt bzw. in seine Warteschlange eingereiht werden können. Wir gehen hier zunächst von periodischen Tasks aus, aperiodische Aktivitäten können jedoch gleichermaßen bedient werden. Wenn diese Tasks nicht zeitkritisch sind, also keine relativen Fris-

ten innerhalb ihrer Perioden besitzen, die einzuhalten wären, dann kann die Warteschlange mit einer FIFO-Strategie abgearbeitet werden und die Tasks würden bestmöglichst (*best effort*) mit der eingestellten Server-Bandbreite bedient werden. Besitzen wiederum die gekapselten Tasks eigene Fristen, so muss nun sichergestellt werden, dass sie diese auch bei einer server-gesteuerten Ausführung rechtzeitig schaffen. Dafür müssen zunächst folgende Anforderungen erfüllt werden:

1. Es muss eine spezielle Disziplin für die warteschlangen-interne Verwaltung unterschiedlicher Task-Jobs, also ein CBS-interner Scheduling-Mechanismus, gewählt werden.
2. Der Server muss im Sinne seiner Kapazität $R_{i,j}$ und Periode $P_{i,j}$ so dimensioniert werden, dass die Tasks genügend Rechenzeit bekommen und ihre Fristen einhalten.
3. Die Planbarkeit der Tasks innerhalb der etablierten Server-Bandbreite muss durch eine lokale Planbarkeitsanalyse nachgewiesen werden.

Die Lösungsansätze für die oben genannten Anforderungen, die für das in entwickelte Scheduling-Modell von ARTOS gewählt wurden, werden nun im Folgenden näher erläutert.

CBS-internes Scheduling

Die CBS-interne Verwaltung und Aktivierung der gekapselten Tasks entspricht dem Scheduling auf der Task-Ebene im vorgestellten hierarchischen Modell. Es ist klar, dass hier ein geeigneter Scheduling-Mechanismus gewählt werden muss, welcher die Echtzeitanforderungen der Tasks im Sinne ihrer Fristen berücksichtigt. In Kapitel 3.2.1 wurde bereits argumentiert, dass hierfür prinzipiell ein beliebiger Algorithmus aus der Echtzeittheorie gewählt werden kann (z.B. RMS, EDF, LLF). Theoretisch kann ebenfalls für jede Anwendung ein unterschiedlicher Mechanismus auf der Task-Ebene greifen. Um den Aufwand für die lokale Planbarkeitsanalyse für alle Anwendungen konsistent zu halten und die Implementierung des Modells zu erleichtern wurde im Rahmen dieser Arbeit für alle Anwendungen Earliest Deadline First als Scheduling-Mechanismus auf der Task-Ebene (und somit als CBS-interner Mechanismus) gewählt (vgl. Kapitel 3.2.1). Die darauf basierende Planbarkeitsanalyse und Server-Dimensionierung werden im Folgenden vorgestellt.

Obwohl der klassische CBS-Ansatz eine nicht-präemptive Disziplin für die CBS-interne Verwaltung der Task-Jobs vorschreibt (vgl. Punkt 6 der CBS-Definition aus Seite 89) spricht aus der Sicht eines Task-Schedulers im vorgestellten Modell nichts gegen eine präemptive Strategie. Diese Anforderung entstammt ursprünglich aus der Notwendigkeit der Sequentialisierung der Jobs

der bedienten Task. Damit wird verhindert, dass Ausführungsinstanzen ein und derselben Task parallel ausgeführt werden, sich gegenseitig verdrängen oder sogar überholen können.

Im Gegensatz dazu verwaltet die interne Warteschlange eines CBS im hier vorgestellten Modell unterschiedliche Tasks. Darüber hinaus wird vorausgesetzt, dass ein sukzessiver Job einer Task nicht aktiviert werden kann, bevor ihr letzter aktiver Job vollständig ausgeführt oder eben abgebrochen wurde. Dieses Modell entspricht eher der generellen Beschaffenheit einer möglichen Ausführungsplattform, in der Anwendungsaktivitäten durch Software-Threads auf der Ebene eines Echtzeit-Betriebssystems abgebildet werden⁷. Somit verweilt stets nur eine Ausführungsinstanz einer Task innerhalb eines Task-Schedulers (und damit auch innerhalb der Warteschlange eines CBS). Die Warteschlange des CBS entspricht dann der Warteschlange (*Run-Queue*) eines Schedulers innerhalb eines gängigen Betriebssystems. Verdrängungen können nun dann auftreten, wenn zum Beispiel der eintreffende Job einer Task eine kürzere absolute Frist besitzt als der aktuelle Job der gerade ausgeführten Task. Die verdrängte Task bleibt lauffähig und wird bei EDF kurzfristig wieder ganz vorn in die Warteschlange eingereiht, bis sie wieder dran kommt.

Beispiel von zwei Tasks innerhalb eines CBS

Im Folgenden wird anhand eines Beispiels das Scheduling von zwei Tasks innerhalb eines Constant Bandwidth Servers untersucht. Angenommen τ_1 und τ_2 gehören zur selben Anwendung, die von einem CBS gekapselt wird und neben einer anderen Task τ_3 mit harten Echtzeitanforderungen ausgeführt wird. Tabelle 3.1 zeigt die Definitionen der Tasks bezüglich ihrer Perioden, Kosten und Deadlines.

	Periode (T_i)	Kosten (C_i)	Deadline (D_i)
τ_1	5	2	4
τ_2	3	1	2
τ_3	5	1	5

Tabelle 3.1: Task-Definitionen

	CBS	τ_3
Kapazität R_s	2.4	
Periode P_s	3	
Bandbreite B_i	0.8	0.2

Tabelle 3.2: Bandbreiten

Tabelle 3.2 zeigt die Konfiguration des CBS von τ_1 und τ_2 , sowie die Bandbreitenallokationen im System, für den Server und für die separat ausgeführte Task τ_3 . Der CBS wird mit einer Rechenzeitkapazität von $R_s = 2.4$ Zeiteinheiten (ZE) pro Server-Periode $P_s = 3$ ZE initialisiert. Die allozierte Bandbreite für den Server beträgt somit $B_1 = \frac{R_s}{P_s} = 0.8$. Task τ_3 ist in diesem Beispiel nicht innerhalb eines Servers gekapselt und somit nicht temporal isoliert. Die von ihr zur

⁷Mehr dazu in Kapitel 4.3.2

Ausführung benötigte Bandbreite ergibt sich aus ihrer CPU-Auslastung⁸ $U_3 = \frac{C_3}{T_3} = 0.2 = B_2$. Solange die Kosten von τ_3 konstant bleiben, ist nach Gleichung 3.18 die Planbarkeit des Systems anhand der allozierten Bandbreiten mit $0.8 + 0.2 \leq 1$ zunächst erfüllt, das System ist dabei voll ausgelastet. Die Kosten von τ_1 und τ_2 können zwar auch variieren, sie werden jedoch durch ihren CBS immer mit der konstanten Bandbreite von $B_s = 0.8$ bedient und innerhalb jeder Server-Periode beschränkt. Würden dagegen die Kosten von τ_3 schwanken, so können dadurch τ_1 und τ_2 beeinflusst werden und ihre Fristen verpassen, da τ_3 nicht temporal isoliert ist. Wir nehmen jedoch zunächst an, dass die Kosten von τ_3 konstant bleiben. Die Frage, die sich nun ergibt, ist, ob τ_1 und τ_2 , mit der aktuellen Konfiguration ihres Servers neben τ_3 , bei voller Auslastung des Systems, durch das Zusammenspiel von CBS- und internem EDF-Scheduling ihre Fristen einhalten können.

Abbildung 3.13: Beispiel von zwei Tasks mit einem CBS

Abbildung 3.13 veranschaulicht die Aktivierungsreihenfolge der Tasks innerhalb ihrer Hyperperiode, sowie die Konfiguration des Servers zu jedem Zeitpunkt t mit seiner aktuellen Kapazität c_s und Deadline D_{s_i} . Zum Zeitpunkt $t = 0$ starten alle Tasks τ_1 , τ_2 und τ_3 synchron; dies ist bekanntlich der schlimmste Fall bzgl. der Systemauslastung [105]. Mit der Aufweckregel wird die

⁸da in diesem Beispiel die Task-Deadline gleich der Task-Periode ($T = D$) ist kann hier über die Auslastung gerechnet werden (vgl. Kapitel 2.1.2)

Kapazität des Servers auf $c_s = R_s = 2.4$ aufgeladen und seine Scheduling-Deadline $D_{s_1} = 3$ gesetzt. Der CBS hat somit eine kleinere Deadline als τ_3 und wird aktiviert. Innerhalb des Servers hat die Task τ_2 eine kürzere Frist als τ_1 und wird somit zuerst ausgeführt. Task τ_2 ist ebenfalls bereit und befindet sich derweil in der Warteschlange des Servers. Zum Zeitpunkt $t = 1$ terminiert der erste Job von τ_1 und τ_2 wird mit der aktuellen Kapazität $c_s = 1.4$ und Deadline $D_1 = 3$ des Servers bedient (CBS-Regel 8). Zum Zeitpunkt $t = 2.4$ ist die Kapazität aufgebraucht; sie wird auf $c_s = 2.4$ vollständig aufgeladen und die Server-Deadline um eine Periode nach hinten verschoben $D_{s_2} = D_{s_1} + P_s = 6$ (CBS-Regel 4). Somit hat nun τ_3 eine kürzere Deadline $D_3 = 5$ als der Server und wird für 1 Zeiteinheit (ZE) aktiviert – der Server wird also kurzfristig verdrängt. Wenn der Job von τ_3 terminiert wird der Rest von τ_1 bis $t = 4$ zu Ende ausgeführt, sodass sie ihre Deadline einhalten kann. Zum selben Zeitpunkt wird der nächste Job von τ_2 aktiviert, der Server bleibt somit aktiv und τ_2 wird mit seiner aktuellen Kapazität $c_s = 1.8$ und Deadline D_{s_2} bedient. Zum Zeitpunkt $t = 5.8$ ist die Kapazität des Servers erneut aufgebraucht. Sie wird wieder voll aufgeladen und die Deadline um eine Periode auf $D_{s_3} = 9$ verlängert. Die Server-Deadline ist damit immer noch kleiner als die von τ_3 , somit wird der Server mit seinen internen Tasks weiter ausgeführt. Zum Zeitpunkt $t = 8$ sind alle Jobs von τ_1 und τ_2 vollständig terminiert und der Server bleibt für 1 ZE inaktiv mit einer Rest-Kapazität von $c_s = 0.2$. In diesem Zeitintervall wird der zweite Job von τ_3 ausgeführt. Wenn bei Zeitpunkt $t = 9$ der nächste Job von τ_2 eintrifft ist $c_s = 0.2 \geq (D_{s_3} - r) \cdot B = (9 - 9) \cdot 0.8 = 0$, also wird mit der Aufweckregel (CBS-Regel 7) c_s wieder vollständig aufgeladen und eine neue Server-Deadline $D_{s_4} = 9 + P_s = 12$ gesetzt. Da der nächste Job von τ_3 noch nicht aktiv ist, bleibt der CBS mit seinen Tasks in Ausführung, bis im Zeitpunkt $t = 11.4$ seine Kapazität wieder aufgebraucht ist. Der Vorgang von Regel 4 wird wiederholt, die Kapazität voll aufgeladen und die Server-Deadline wieder um eine Server-Periode verlängert $D_{s_5} = 15$. Zum Zeitpunkt $t = 14$ sind alle Task-Jobs terminiert und der CBS mit einer Restkapazität von $c_s = 0.8$ inaktiv. Zu Beginn der neuen Hyperperiode in $t = 15$ wird die Server-Kapazität wieder voll aufgefüllt und eine neue Server-Deadline mit $D_{s_6} = 15 + 3 = 18$ gesetzt, da hier mit $D_{s_5} - r = 0$ die Aufweckregel erneut greift. Da in diesem Beispiel die Periode der Task τ_3 mit 5 ZE ein Teiler der Hyperperiode H von τ_1 und τ_2 ist und die Serverkonfiguration zum Zeitpunkt $t = 15$ die selbe ist wie zum Zeitpunkt $t = 0$, wiederholt sich die Aktivierungsreihenfolge der drei Tasks innerhalb der nächsten und somit auch innerhalb jeder Hyperperiode. Dabei schaffen τ_1 und τ_2 stets ihre eigenen Fristen innerhalb des Servers.

Implikationen für die Regeln und die Konfiguration von CBS

Für die zuvor auf Seite 89 vorgestellten CBS-Regeln hat die Ausführung mehrerer Tasks innerhalb eines Servers folgende Implikationen:

- Regel 5 wird für *Jobs mehrerer Tasks* erweitert, d.h. ein Server ist *inaktiv*, wenn für alle gekapselten Tasks keine ausstehenden Jobs existieren und seine Warteschlange leer ist.
- Dem Job *einer jeden Task* wird die aktuelle Server-Deadline zugewiesen (Regel 2 und 9).
- Regel 6 wird um *präemptives lokales EDF-Scheduling* erweitert, d.h. wenn ein neuer Job einer Task eintrifft, wird er entsprechend seiner sich zum Aktivierungszeitpunkt etablierenden absoluten Deadline in die server-interne Warteschlange nach EDF eingereiht. Unter Umständen kann aber auch der gerade aktive Job einer Task verdrängt werden, wobei er (nach EDF) temporär an den Anfang der Warteschlange zurückgestellt wird.

Die anderen Regeln 1, 3, 4, 7 und 8 bleiben auch für mehrere Tasks zunächst unverändert.

Im vorherigen Beispiel wurde durch die Bandbreiten-Belegung im System sichergestellt, dass alle Tasks ihre Anforderungen erfüllen können. Dafür wurde für τ_1 und τ_2 die benötigte Server-Bandbreite im Voraus berechnet. Obwohl aus globaler Sicht jeder durch den CBS bediente Task-Job mit der aktuellen, virtuellen Frist des Servers D_{s_i} aktiviert wurde (CBS-Regel 2), war der Server so dimensioniert, dass die Jobs beider Tasks genügend Rechenzeit innerhalb jeder Periode bekommen haben, um rechtzeitig vor ihrer jeweiligen eigenen First zu terminieren. Dies wurde durch die Herstellung einer Beziehung zwischen der allozierten Server-Bandbreite, genauer durch die Parameterisierung des Servers mit einer Kapazität und Periode, und der Planbarkeit seiner internen Task-Menge erreicht.

Im vorgestellten hierarchischen Scheduling-Modell wird diese Beziehung zur Laufzeit an definierten Stellen periodisch überprüft und hergestellt. Die allozierte Bandbreite für eine Anwendung wird dabei notfalls neu kalibriert. Diese Stellen wurden im vorherigen Kapitel 3.2.1 bereits vorgestellt – es sind die Zeitpunkte an denen in definierten Abständen die Gültigkeit der Approximation einer jeden Task überprüft und ihre Kostenschätzung bei Bedarf aktualisiert wird. Eine Bandbreiten-Aktualisierung für ein CBS findet also auf der Applikations-Ebene nur dann statt, wenn sich die Kostenschätzung einer internen Anwendungs-Task ändert. Falls nicht, bleibt die letzte Konfiguration des Servers bis zum nächsten Überprüfungspunkt bestehen.

Die Beziehung zwischen Bandbreite und Planbarkeit der server-internen Task-Menge wird nun im Folgenden näher vorgestellt.

Dimensionierung eines CBS und lokale Planbarkeitsanalyse

Die grundsätzliche Frage ist nun, wie muss der Server einer Anwendung entsprechend seiner *Periode* $P_{i,j}$ und *Kapazität* $R_{i,j}$ dimensioniert werden, damit die von ihm gekapselten Anwendungstasks im Rahmen des CBS-Schedulings ihre Fristen rechtzeitig schaffen. Die Punkte 2 und 3, der auf Seite 91 präsentierten Anforderungen für die Kapselung mehrerer Tasks innerhalb eines CBS, hängen also unmittelbar zusammen. Der entsprechende Lösungsansatz, welcher hierfür im Rahmen von ARTOS gewählt wurde, geht jedoch einen inversen Weg, bei dem gerade durch eine lokale Planbarkeitsanalyse, die für eine erfolgreiche Ausführung einer CBS-internen Task-Menge benötigte Server-Kapazität für eine entsprechende Server-Periode gewählt wird. Dieser Vorgang wird im Folgenden erklärt.

Wahl der Server-Periode $P_{i,j}$:

Um eine fehlerfreie Ausführung der periodischen Task $\tau_{i,k} \in \mathcal{T}_i$ mit der höchsten Frequenz innerhalb einer Anwendung A_i zu garantieren, muss der Server mit mindestens derselben Frequenz arbeiten. Nur auf diese Weise können die Fristen aller Anwendungs-Tasks durch die CBS-Aktivierung berücksichtigt werden.

Betrachten wir nun als Gegenbeispiel dazu die Konfiguration von zwei Applikationen A_1 und A_2 , gekapselt durch zwei CBS, und ihrer internen Tasks $\tau_{i,k} = (T_{i,k}, C_{i,k}, D_{i,k})$ mit entsprechenden Perioden $T_{i,k}$, Kosten $C_{i,k}$ und Fristen $D_{i,k}$:

$$A_1 = \{\tau_{1,k} \in \mathcal{T}_1 \mid \tau_1 = (5, 1, 5), \tau_2 = (3, 1, 3)\}$$

$$A_2 = \{\tau_{2,k} \in \mathcal{T}_2 \mid \tau_3 = (5, 1, 5), \tau_4 = (10, 2, 10)\}$$

Da hier Task-Fristen gleich den Task-Perioden gewählt wurden, ergeben sich die entsprechenden Server-Bandbreiten der Anwendungen über die CPU-Auslastung ihrer internen Tasks mit $U_i = \sum_k \frac{C_{i,k}}{T_{i,k}} = B_i$ als $B_1 = 1/5 + 1/3 = 8/15 = 0.5\bar{3}$ und $U_2 = B_2 = 1/5 + 2/10 = 2/5 = 0.4$. Der Auslastungstest ist bei $D_{i,k} = T_{i,k}$ bekanntlich bereits hinreichend für den Nachweis der Planbarkeit einer Task-Menge (siehe Kapitel 2.1). Nach Gleichung 3.18 ist auch die Gesamtkonfiguration des Systems anhand der etablierten Server-Bandbreiten planbar.

Da die Bandbreiten durch unterschiedliche Konfigurationen von Server-Kapazitäten und Server-Perioden eingestellt werden können, wählen wir zunächst exemplarisch die Server-Perioden gleich den Task-Hyperperioden der jeweiligen Anwendungen, d.h. $P_1 = 15$ und $P_2 = 10$. Durch die CBS-Definition $B_i = R_i/P_i$ ergeben sich dann die Server-Kapazitäten $R_1 = P_1 \cdot B_1 = 15 \cdot 0.5\bar{3} = 8$ ZE und $R_2 = P_2 \cdot B_2 = 10 \cdot 0.4 = 4$ ZE. Für die Laufzeitumgebung der Anwendungen werden nun zusätzlich folgende Bedingungen aufgestellt:

1. *Synchroner Start:* Da die genauen Aktivierungszeiten der Anwendungen innerhalb einer ARTOS-Plattform nicht bekannt sind, wird ein synchroner Start aller Applikationen und ihrer internen Tasks angenommen. Damit wird die Planbarkeit der Anwendungen mit ihren CBS für den schlimmsten möglichen Fall untersucht [105].
2. *Durchsetzung von Bandbreiten:* Wenn ein Server mit der kürzesten Deadline vorübergehend *inaktiv* wird, werden keine Tasks anderer Server ausgeführt. Somit okkupiert ein Server allein die CPU, solange er die kürzeste Frist besitzt. Auf diese Weise wird überprüft, ob die Planbarkeit der Tasks allein durch die Dimensionierung der Server mit den entsprechenden Kapazitäten und Perioden und ihrer strikten EDF-basierten Aktivierung eingehalten wird.
3. *Abbruch bei Fristverletzungen:* Wenn eine Task ihre aktuelle Frist verpasst, wird ihr Job unverzüglich abgebrochen. Um harten Echtzeitanforderungen näher heranzukommen wird hier angenommen, dass ein Ergebnis ausgeliefert nach seiner Frist keinen Nutzen mehr hat. Dementsprechend werden auch Jobs, die bis zu ihrer absoluten Frist gar nicht ausgeführt wurden, ausgelassen und nicht rückwirkend aktiviert.

Abbildung 3.14: Anwendungen mit Server-Perioden gleich den Task-Hyperperioden

Die Aktivierungsreihenfolge der Tasks mit den entsprechenden Server-Konfigurationen ist in Abbildung 3.14 dargestellt⁹. Da der CBS von A_2 eine kürzere Deadline $D_{2_1} = 10$ als der von

⁹Die Kapazitäten der Server wurden hier aus Übersichtsgründen nicht dargestellt.

A_1 ($D_{1_1} = 15$) hat, wird A_2 zuerst ausgeführt. Zum Zeitpunkt $t = 6$ ist die Server-Kapazität von A_2 aufgebraucht, also wird $c_2 = 4$ neu aufgeladen und eine neue Deadline $D_{2_2} = 20$ gesetzt. Hier wird A_1 mit Deadline $D_{1_1} = 15$ und der noch vollen Kapazität $c_1 = 8$ aktiv. Zu diesem Zeitpunkt hat A_1 jedoch bereits drei Fristverletzungen erlitten – man beachte, dass auch ohne die Bedingung 2 für inaktive Server (siehe Seite 97) τ_2 zum Zeitpunkt $t = 3$ ihre Frist trotzdem verpassen würde. Wenn zum Zeitpunkt $t = 15$ die ersten Jobs von τ_1 und τ_2 innerhalb der neuen Hyperperiode eintreffen, ist ihr Server gerade inaktiv und hat noch eine Restkapazität von $c_1 = 3$. Hier greift nun die CBS-Regel 7 (siehe Seite 89) mit $c_1 = 3 \geq (D_i - r_i) \cdot B_i = 0$, also wird $c_1 = 8$ aufgeladen und eine neue Server-Deadline $D_{1_2} = r_2 + P_2 = 15 + 15 = 30$ gesetzt. Zu dieser Zeit hat Anwendung A_2 bereits auch eine Fristverletzung erlitten und wird nun mit der zum Zeitpunkt $t = 6$ gesetzten Kapazität $c_2 = 4$ und Deadline $D_{2_2} = 20$ aktiviert. Dies führt wiederum zu weiteren Fristverletzungen innerhalb A_1 . Obwohl die Planbarkeit auf der Server-Ebene mit $\sum_{i=1}^n B_i = 0.5\bar{3} + 0.4 = 0.9\bar{3} \leq 1$ gewährleistet ist, ist die Server-Konfiguration nicht optimal für die Einhaltung der Fristen aller Anwendungs-Tasks.

Nun wird dasselbe Anwendungsszenario mit einer anderen Server-Konfiguration abgespielt, bei der die Serverperioden mit $P_i = \min(T_{i,k})$ gleich der kleinsten Task-Periode der jeweiligen Anwendung gewählt werden, also für A_1 ist $P_1 = 3$ ZE und für A_2 ist $P_2 = 5$ ZE. Da durch die Task-Konfigurationen die partiellen CPU-Auslastungen der Anwendungen mit $U_1 = 8/15 = 0.5\bar{3}$ und $U_2 = 2/5 = 0.4$ unverändert bleiben, bleiben auch die allozierten Bandbreiten der Server identisch zum vorherigen Beispiel. Mit den neuen Server-Perioden ergeben sich dann die Server-Kapazitäten $R_1 = P_1 \cdot B_1 = 3 \cdot 0.5\bar{3} = 1.6$ und $R_2 = P_2 \cdot B_2 = 5 \cdot 0.4 = 2$ ZE. Die Aktivierungsreihenfolge der Anwendungen mit den neuen Server-Konfigurationen ist in Abbildung 3.15 dargestellt.

In dieser Variante treten keine Fristverletzungen mehr auf. Das Ergebnis kann folgendermaßen interpretiert werden:

- Durch die Dimensionierung eines Servers über die kleinste Periode $P_i = \min(T_{i,k})$ einer Task $\tau_{i,min}$ innerhalb einer Anwendung, wird seine verfügbare Bandbreite über die Zeit in einer Kapazität c_s pro Server-Periode P_i zerstückelt. Die restliche Kapazität $P_i - c_s$ steht allen anderen Anwendungen im System zur Verfügung und wird innerhalb ihrer eigenen Server-Bandbreiten berücksichtigt.
- Diese Kapazität $c_s = c_g + c_r$ besteht aus einem Grundanteil c_g für die Task $\tau_{i,min}$ und einem Restfragment c_r für alle anderen Anwendungs-Tasks $\tau_{i,k}$ mit größeren Perioden.
- Der Grundanteil $c_g = C_{i,min}$ garantiert die vollständige Abarbeitung der Task $\tau_{i,min}$ innerhalb jeder Server-Periode und somit auch innerhalb ihrer eigenen Periode und Frist.

Abbildung 3.15: Anwendungen mit Server-Perioden gleich den kleinsten Task-Perioden

- Das Restfragment c_r besteht aus der Summe der anteiligen Anforderungen der restlichen Anwendungs-Tasks bezüglich ihrer eigenen Perioden (und Fristen) im Verhältnis zur gewählten Server-Periode $P_i = \min(T_{i,k})$, also aus $\sum_{i=1}^k \frac{P_i \cdot C_{i,k}}{T_{i,k}} - C_{i,\min} = c_r$
- Je nach dem wie die anwendungs-interne Task-Aktivierungen und die Verdrängungen der Server untereinander sich zur Laufzeit etablieren, kann der aktuelle Job von $\tau_{i,\min}$ oder einer anderen Task derselben Anwendung zwischen mehreren Server-Perioden aufgeteilt werden (siehe z.B. Abbildung 3.15). Die Kapazität c_s pro Server-Periode bleibt jedoch dabei zugesichert und garantiert mit $c_s = c_g + c_r$ das Erreichen aller Task-Fristen bei unveränderten Kosten.

Beispiel:

Betrachten wir den Server der Applikation A_1 aus dem vorherigen Beispiel in Abbildung 3.15.

Die allozierte Bandbreite des Servers ist $B_1 = 0.5\bar{3}$ und mit $P_1 = \min(T_{1,k}) = T_{1,2} = 3$ ergibt sich die Server-Kapazität $R_1 = P_1 \cdot B_1 = 3 \cdot 0.5\bar{3} = 1.6$. Innerhalb jeder Periode bekommt also der Server ein Budget von $c_s = c_g + c_r = C_{1,2} + c_r$ also von $1.6 = 1 + c_r$. Da A_1 aus nur einer weiteren Task τ_1 mit einer größeren Periode von $T_{1,1} = 5$ und Kosten $C_{1,1} = 1$ besteht, muss also innerhalb aller $T_{1,1}/P_1 = 1.6$ Server-Perioden eine zusätzliche Kapazität von $C_{1,1} = 1$ ZE für τ_2 vorhanden sein. Der Anteil dieser Kapazität innerhalb jeder Server-Periode lässt sich einfach durch die Verhältnisgleichung $T_{1,1} : C_{1,1} = P_1 : c_r$ mit $c_r = \frac{P_1 \cdot C_{1,1}}{T_{1,1}} = \frac{1 \cdot 3}{5} = 0.6$ errechnen. Wie man in Abbildung 3.15 sieht, wurden innerhalb A_1 mehrere Jobs von τ_1 über zwei Server-Perioden aufgeteilt (z.B. bei $t = 1.6, 10.4, \dots$). Ebenso wurde ein Job von τ_2 aufgrund einer Verdrängung ihres Servers auch mehrmals verdrängt (z.B. bei $t = 6.8$ und $t = 21.8$). Trotzdem halten beide Tasks durch die allozierte Server-Kapazität pro Periode ihre Fristen ein.

Durch diese Variante der Dimensionierung wurde also in jeder Server-Periode eine Grundkapazität für die Task mit der kleinsten Periode und eine Restkapazität für alle anderen Anwendungstasks reserviert, sodass alle Anwendungsaktivitäten ihre Fristen innerhalb ihrer Perioden schaffen können. Auf diese Weise wurden ebenfalls genügend Verdrängungspunkte der Server generiert, so dass ihre internen Task-Aktivierungen berücksichtigt werden. Prinzipiell kann auch eine beliebig kleinere Server-Periode $P_i < \min(T_{i,k})$ mit entsprechender Kapazität gewählt werden, diese würde jedoch weitere unnötige Server-Verdrängungen erzeugen und somit den Scheduling-Overhead im System erhöhen.

Im letzten Beispiel konnte die benötigte Server-Bandbreite für eine Anwendung einfach aus der Summe der partiellen Auslastungen ihrer Tasks $B_i = U_i = \sum_{k=1}^n \frac{C_{i,k}}{T_{i,k}}$ berechnet werden, da die Fristen der Tasks gleich ihren Perioden ($D_{i,k} = T_{i,k}$) gesetzt wurden. Nach Gleichung 2.1 auf Seite 23 ist bei $D_{i,k} = T_{i,k}$ die Planbarkeit einer Task-Menge über die notwendige Bedingung $U = \sum_{i=1}^n \frac{C_i}{T_i}$ bereits erwiesen. Sind also in einem System alle Anwendungen innerhalb gleichermaßen dimensionierten Servern temporal isoliert, so kann nun durch $\sum_{i=1}^n B_i$ eine Aussage über die Planbarkeit des Gesamtsystems auf der Task-Ebene gemacht werden. Wenn allerdings eine Task eine kleinere Frist als ihre entsprechende Periode besitzt ($D_{i,k} \leq T_{i,k}$) ist eine aufwendigere lokale Planbarkeitsanalyse für die betroffene Anwendung notwendig, um die benötigte Server-Kapazität für sie zu berechnen. Diese wird nun im Folgenden vorgestellt.

Wahl der Server-Kapazität $R_{i,j}$:

Die benötigte Kapazität $R_{i,j}$ für den Constant Bandwidth Server einer Anwendung A_i wird durch eine lokale Planbarkeitsanalyse für ihre internen Tasks $\tau_{i,k} \in \mathcal{T}_i$ ermittelt. Dabei werden die aktuellen Kostenschätzungen $C_{i,k,l}$ der Tasks für den jeweiligen Betriebsmodus der Anwendung verwendet. Diese Kapazität wird dann mit der gewählten Server-Periode $P_{i,j} = \min(T_{i,k,l})$ für den entsprechenden Anwendungsmodus normiert und durch den Server periodisch reserviert. Somit

belegt jede Anwendung eine bestimmte Ressourcenbandbreite in ihrem aktuellen Betriebsmodus¹⁰. Innerhalb dieser Bandbreite wird garantiert, dass die Anwendungs-Tasks genügend Rechenzeit pro Server-Periode bekommen, um ihre Fristen einzuhalten.

Wie bereits erwähnt kann die Planbarkeit einer Task-Menge mit Fristen gleich Perioden $D_{i,k} = T_{i,k}$ über einen einfachen Auslastungstest nachgewiesen werden. Dabei werden über die partiellen Auslastungen der Anwendungen $U_i = \sum_{k=1}^K \frac{C_{i,k}}{T_{i,k}}$ direkt ihre benötigten Bandbreiten B_i ermittelt. Die Server-Kapazitäten ergeben sich dann, wie im Beispiel auf Seite 100, mit den gewählten Server-Perioden durch $R_i = B_i \cdot P_i$. Wenn jedoch die Frist einer einzigen Task kleiner ist als ihre Periode $D_{i,k} < T_{i,k}$, dann muss für die betroffene Anwendung eine Planbarkeitsanalyse über die Rechenzeitanforderung (Processor Demand) ihrer Tasks betrieben werden, um ihre Server-Kapazität zu ermitteln. Die Server-Kapazität wird also aus dem kumulativen Processor Demand der Anwendung ermittelt.

Wie in Kapitel 2.1.1 bereits erklärt wurde, muss bei der Planbarkeitsanalyse für Earliest Deadline First an allen Deadline-Punkten innerhalb der Task-Hyperperiode überprüft werden, ob der aktuelle Wert der Rechenzeitanforderungsfunktion $h(t)$ (siehe Gleichung 2.3) die maximal zulässige Anforderung $h(t) = t$ übersteigt (siehe Abbildung 2.6). Wenn dies nicht der Fall ist, ist die Task-Menge planbar. Da bei manchen Task-Perioden die Hyperperiode relativ groß ausfallen kann, kann die Anzahl der benötigten Tests z.B. durch die von Baruah et al. vorgeschlagene Approximation der Rechenzeitanforderungsfunktion mit einer Gerade $c_D(t)$ bis zu einem Zeitpunkt t' eingeschränkt werden (siehe Abbildung 2.7). An diesem Punkt schneidet die Approximationsgerade die maximale Rechenzeitanforderung $h(t) = t$ und verbleibt stets unterhalb dieser, wodurch die Planbarkeit der Task-Menge nachgewiesen ist.

Die Grundidee für die Ermittlung der Serverkapazität ist also, einen repräsentativen Wert für die kumulative Rechenzeitanforderung der Tasks einer Anwendung über die Zeit für die gewählte Server-Periode $P_i = \min(T_{i,k})$ zu bestimmen und zwar so, dass alle Tasks ihre Fristen schaffen. Hier wird wieder die Eigenschaft von EDF ausgenutzt, dass sich bei gleich bleibenden Kosten die Aktivierungsreihenfolge der Tasks in jeder Hyperperiode wiederholt. Außerdem wird wieder von einem synchronen Start ausgegangen, um alle möglichen Aktivierungsphasen der Tasks zu berücksichtigen. Diese sind ja in einem realen System zunächst nicht bekannt und können später durch unvorhersehbare Modi-Wechsel der Anwendung permutiert werden.

Gesucht ist also der Wert der kumulativen Rechenzeitanforderung $h(H)$ zum Zeitpunkt der Task-Hyperperiode. Da für die Berechnung der Approximationsgerade $c_D(t) = t \cdot U_i + U_i \cdot \max(T_{i,k,l} - D_{i,k,l})$ ein geringerer Aufwand als für die Berechnung Rechenzeitanforderung der Tasks anfällt,

¹⁰Für den Betriebsmodus Null einer jeden Anwendung ist die Bandbreite gleich Null.

kann stattdessen auch der Approximationswert von $c_D(H)$ verwendet werden. Nachdem die Gerade $c_D(t)$ die DBF approximiert, kann mit ihrem Wert zum Zeitpunkt der Hyperperiode $c_D(H)$ ein Schätzwert für die benötigte Rechenzeitkapazität der Tasks innerhalb der gesamten Hyperperiode bestimmt werden (siehe Abbildung 3.16). Die Konstanten der Geradengleichung U_i (partielle Auslastung der Anwendung) und $\max(T_{i,k,l} - D_{i,k,l})$ (der maximale Abstand zwischen der Periode und der Frist einer Task) wurden bereits bei der Überprüfung der notwendigen Planbarkeitsbedingung $U_i = \sum_{k=1}^K \frac{C_{i,k}}{T_{i,k}} \leq 1$ und bei der Bestimmung von $t' = \frac{U_i}{1-U_i} \cdot \max(T_{i,k,l} - D_{i,k,l})$ (entsprechend der Gleichung 2.7 auf Seite 27) ausgerechnet. Der restliche Aufwand für die Bestimmung der Server-Kapazität beläuft sich somit auf eine einfache Multiplikation und einer Addition innerhalb der Geradengleichung. Die kumulative Rechenzeitanforderung der Tasks wird dann nur für die Planbarkeitstests innerhalb des Intervalls $[0..t']$ ausgerechnet.

Abbildung 3.16: Beispiel für $c_D(H)$ für die Task-Menge aus Abbildung 2.6

Der Wert $c_D(H)$ ist also eine Approximation der benötigten Rechenzeit innerhalb jeder Hyperperiode, mit der alle Tasks ihre Fristen erreichen können. Dieser Wert steht zunächst im Verhältnis zur Hyperperiode der Tasks und wird darum mit der gewählten Server-Periode P_i normiert. Die Konfiguration der Kapazität eines Constant Bandwidth Servers ist also gegeben durch:

$$\begin{aligned}
 R_i &= P_i \cdot \frac{c_D(H_i)}{H_i} \Rightarrow P_i \cdot \frac{(H_i \cdot U_i + U_i \cdot \max(T_{i,k,l} - D_{i,k,l}))}{H_i} \Rightarrow \\
 R_i &= P_i \cdot U_i \cdot \left(1 + \frac{\max(T_{i,k,l} - D_{i,k,l})}{H_i}\right)
 \end{aligned} \tag{3.20}$$

In Gleichung 3.20 wird die Task-Hyperperiode H_i in die Gleichung (2.6) der Geraden $c_D(t)$ ein-

der Periode $P_s = 3$, der Kapazität $R_s = 2.4$ und eine Bandbreite $B_s = 0.8$ dimensioniert. Das bedeutet, dass der Server innerhalb einer jeden Periode einen Anstieg der Rechenzeitanforderung von höchstens 2.4 ZE fehlerfrei bedienen kann. In Abbildung 3.17 ist der kumulative Anstieg der Rechenzeitanforderung der Tasks (DBF) und der reservierten Kapazität ($\sum R_s$) am Ende einer jeden Server-Periode dargestellt. Somit hat z.B. die Task-Menge am Ende der zweiten Server-Periode eine Anforderung von $DBF = 4$ ZE erzeugt, während durch den Server bislang $\sum R_s = 4.8$ ZE reserviert wurden. Eine Bedingung für die Planbarkeit der Task-Menge ist also, dass das Verhältnis zwischen der Rechenzeitanforderung und der kumulativ bereitgestellten Rechenzeitkapazität innerhalb jeder Server-Periode kleiner-gleich 1 bleibt, also $\frac{DBF(t)}{\sum_t R_s} \leq 1$. Wie man in Abbildung 3.17 sieht übersteigt dort die kumulative Rechenzeitanforderung der Tasks in keiner Server-Periode die durch den Server garantierte kumulative Kapazität, was die Einhaltung der Taskfristen garantiert.

Aufgrund der Kapazitätsbegrenzung mit $R_s = 2.4$ ZE pro Server-Periode $P_s = 3$, werden bei der tatsächlichen CBS-basierten Aktivierung Teile der Task-Kosten innerhalb einer Server-Periode auf die Folge-Periode verlagert (siehe Abbildung 3.17). Diese Anteile werden bei diesem Beispiel jedoch höchstens um 0.6 ZE innerhalb jeder Server-Periode verzögert, da die verbleibende Restbandbreite $B_{rest} = 1 - B_s = 0.6/3 = 0.2$ beträgt. Somit wird bei der ersten Periode ein Restteil von τ_1 gleich 0.6 ZE auf die zweite Server-Periode verschoben, während dort wiederum ein Restteil von 0.2 ZE für die dritte Periode übrig bleibt usw. Entscheidend für die Einhaltung der Task-Fristen ist dabei jedoch die Abdeckung der Rechenzeitanforderung der Tasks innerhalb jeder Server-Periode mit genügend Server-Kapazität, unabhängig davon wie genau die Task-Kosten durch das CBS-Scheduling zwischen mehreren Server-Perioden aufgeteilt werden. Die Durchsetzung der reservierten Bandbreiten durch das CBS-Scheduling und die EDF-basierte Aktivierung auf der Task-Ebene sorgen dann automatisch für die richtige Ausführungsreihenfolge der Tasks und somit für die Einhaltung ihrer Fristen.

3.2.3 Scheduling auf der System-Ebene

Wie im letzten Kapitel gezeigt wurde, wird mit der Wahl einer entsprechenden Server-Periode und -Kapazität auf der Applikations-Ebene für jede Anwendung dynamisch eine geeignete Ressourcenbandbreite berechnet und reserviert, mit der die Planbarkeit ihrer Aktivitäten in ihrem jeweiligen Betriebsmodus gewährleistet ist. Das Scheduling anhand von Constant Bandwidth Servern sorgt dann automatisch für die Einhaltung der reservierten Bandbreiten, während diese auf der Task-Ebene unter den jeweiligen Anwendungsaktivitäten verteilt werden. Dieses Vorgehen bringt einige Vorteile gegenüber einem flachen Modell, bei dem sämtliche Anwendungs-Tasks

innerhalb eines Systems vom einem globalen Task-Scheduler gesteuert werden:

- Die Planbarkeit des Gesamtsystems auf der Task-Ebene, in seiner aktuellen Konfiguration, kann viel effizienter auf einer höheren Ebene anhand der Bandbreiten der Anwendungen überprüft werden.
- Bei einer Änderung der Kostenschätzung einer Task ändert sich auch die Rechenzeitanforderung und somit auch die Bandbreite ihrer gesamten Anwendung. Eine Planbarkeitsanalyse auf der Task-Ebene muss dann nicht für die gesamte Task-Menge im System betrieben werden, sondern nur für die betroffene Anwendung, um ihre neue Rechenzeitanforderung zu bestimmen (siehe Kapitel 3.2.2).
- Die Auswirkung einer so entstandener Bandbreitenänderung auf die Planbarkeit des Gesamtsystems, kann dann sehr schnell über die Summe der Anwendungs-Bandbreiten für ihre aktuellen Modi mit $\sum_{i=1}^n B_i \leq 1$ festgestellt werden.
- Da die Bandbreiten die (aktuellsten) Ressourcenanforderungen der Anwendungen in ihren unterschiedlichen Modi repräsentieren, kann damit ihre beste und zugleich planbare Modi-Konfiguration mit dem höchsten Nutzen im System ausgewählt werden. Die Suche einer solchen Konfiguration kann somit von der Planbarkeitsanalyse auf der Task-Ebene entkoppelt werden.

Eine Berechnung der optimalen Modi-Konfiguration der Anwendungen ist zunächst dann notwendig, wenn (1) sich die Kostenschätzung einer Task und somit die Bandbreite eine Anwendung ändert, (2) wenn ein Benutzer die Priorität einer Anwendung verändert und die Ressourcen deswegen neu verteilt werden müssen und (3) wenn eine neue Anwendung gestartet wird, oder eine Anwendung terminiert. Der letzte Punkt etabliert automatisch eine Admissionskontrolle für das System. Anwendungen können dabei degradiert bzw. temporär gestoppt werden, um einen potentiellen Engpass zu überwinden oder aber auch in höheren Modi geschaltet werden, um freigeWORDENE Ressourcen für eine bessere Gesamtqualität auszunutzen. Die genaue Vorgehensweise bei der Auswahl der optimalen Modi-Konfiguration wird nun im Folgenden erläutert.

Optimierungsproblem

Die Auswahl der besten Konfiguration, die den Gesamtnutzen im System maximiert, aber auch die verfügbaren Ressourcen optimal auslastet, wird auf ein Optimierungsproblem zurückgeführt:

$$\begin{array}{ll}
\text{Finde eine Selektion von Modi,} & J = \{M_{1,j_1}, M_{2,j_2}, \dots, M_{n,j_n}\}, M_{i,j_i} \in \mathcal{M}_i, \\
\\
\text{maximiere dabei den Nutzen} & \sum_{i=1}^n a_i \cdot U_i(M_{i,j_i}), \\
\\
\text{unter der Bedingung} & \sum_{i=1}^n B_{i,j_i} \leq R \text{ and } |J \cap \mathcal{M}_i| = 1 \ \forall \ \mathcal{M}_i.
\end{array}$$

Tabelle 3.3: Problemdefinition

Die Optimierung selektiert die Anwendungsmodi $M_{1,j_1}, M_{2,j_2}, \dots, M_{n,j_n}$, die den maximalen Nutzen $\sum_{i=1}^n a_i \cdot u_i(M_{i,j_i})$ im System erzeugen, unter der Bedingung, dass eine maximale Ressourcenkapazität R im System mit $\sum_{i=1}^n B_{i,j_i} \leq R$ nicht überschritten wird. Bei der Auswahl der optimalen Konfiguration J können Anwendungen auch temporär in ihren Null-Modi $M_{i,0}$ deaktiviert werden, es findet also eine dynamische Admissionskontrolle statt. Das Optimierungsproblem muss nun zur Laufzeit möglichst effizient gelöst werden.

Die Lösung des Problems, welche im Folgenden präsentiert wird, wurde in Zusammenarbeit mit dem *Institut für Optimierung und Operations Research* an der Universität Ulm entwickelt. Sie basiert auf einem spezialisierten *Rucksackalgorithmus*, welcher mit dynamischer Programmierung implementiert und zur Laufzeit innerhalb einer pseudo-polynomiellen Zeit berechnet werden kann.

Rucksack-basierte Lösung

Unsere rucksack-basierte Lösung ist unabhängig von der Beschaffenheit der Utility-Funktionen U_i der Anwendungen und erzeugt stets eine optimale Lösung, welche innerhalb von einer Algorithmus-Iteration berechnet wird. Bei jedem Iterationsschritt iteriert der Algorithmus über jede gültige Konfiguration von Anwendungsmodi (inklusive der Null-Modi $M_{i,0}$) und wählt eine partielle Lösung mit maximalem Nutzen (oder Qualität) für den nächsten Schritt aus. Die Komplexität vom Algorithmus hängt von einer fest einstellbaren numerischen Auflösung der System-Ressourcen und von zwei dynamischen Faktoren – die aktuelle Anzahl an Applikationen $|A_i|$ und deren Betriebsmodi $|\mathcal{M}_i|$ ab. Bleiben die Werte dieser Faktoren relativ klein, so besitzt der Algorithmus eine pseudo-polynomielle Komplexität und kann sehr effizient zur Laufzeit gelöst werden, wie später in Kapitel 5.6 im Rahmen einer quantitativen Evaluation gezeigt wird. Als nächstes wird die Implementierung mit dynamischer Programmierung vorgestellt.

Für die Berechnung werden zwei Tabellen u und J als Datenstrukturen verwendet.

Für $k \in \{0, 1, \dots, n\}$, $r \in \{0, 1, \dots, R\}$ und $M_{i,j_i} \in \mathcal{M}_i$ soll

$$u(k, r) = a_1 \cdot U_1(M_{1,j_1}) + a_2 \cdot U_2(M_{2,j_2}) + \dots + a_k \cdot U_k(M_{k,j_k}) \quad (3.21)$$

den maximal erreichbaren Nutzen darstellen, so dass $B_{1,j_1} + B_{2,j_2} + \dots + B_{k,j_k} \leq r$ bleibt. Des Weiteren ist

$$J(k, r) = \{M_{1,j_1}, M_{2,j_2}, \dots, M_{k,j_k}\} \quad (3.22)$$

eine optimale Konfiguration, welche die gegebenen Bedingungen erfüllt.

Die Tabelle $u(k, r)$ enthält den maximal möglichen Nutzen der ersten k Applikationen für eine totale Ressourcenbegrenzung r , während in $J(k, r)$ die selektierten Modi gespeichert werden, die im jeweiligen Schritt zur maximalen Qualität geführt haben.

Die Tabellen werden für alle $r = 0, 1, \dots, R$ mit $u(0, r) = 0$ und $J(0, r) = \emptyset$ initialisiert. Für $k \in \{1, 2, \dots, n\}$ und $r \in \{0, 1, \dots, R\}$ gilt dann die folgende Rekursion:

$$u(k, r) = \max_{M_{k,j} \in \mathcal{M}_k} \{u(k-1, r - B_{k,j}) + a_k \cdot U_k(M_{k,j})\} \quad (3.23)$$

während mit

$$J(k, r) = J(k-1, r - B_{k,j}) \cup \{M_{k,j}\} \quad (3.24)$$

eine optimale Selektion gegeben ist. Die Überprüfung, ob $r - B_{k,j}$ positiv ist und sich darum immer noch innerhalb der Tabelle befindet, garantiert die Einhaltung der Bedingung $B_{1,j_1} + B_{2,j_2} + \dots + B_{n,j_n} \leq R$. Diese Definition ermöglicht nun die Formulierung des Algorithmus in Pseudo-Code (siehe Algorithm 1).

Algorithm 1 OPTIMIERUNGSLGORITHMUS

```

1: for  $r = 0 \rightarrow R$  do
2:    $u[0][r] \leftarrow 0, J[0][r] \leftarrow \emptyset$ 
3: end for
4: for  $k = 1 \rightarrow n$  do
5:    $u[k][0] \leftarrow 0$ 
6:   for  $r = 1 \rightarrow R$  do
7:      $max \leftarrow 0$ 
8:     for  $M_{k,j} \in \mathcal{M}_k$  do
9:        $b \leftarrow u[k-1][r - R_{k,j}] + a_k \cdot U_k(M_{k,j})$ 
10:      if  $b > max$  then
11:         $max \leftarrow b$ 
12:         $J[k][r] \leftarrow J[k-1][r - R_{k,j}] \cup \{M_{k,j}\}$ 
13:      end if
14:    end for
15:     $u[k][r] \leftarrow max$ 
16:  end for
17: end for
18: return  $J[n][R]$ 

```

Beispiel:

Es sei die folgende Konfiguration von drei Anwendungen mit entsprechenden Bandbreiten für ihre Modi gegeben:

$$\begin{aligned}
A_1 : B_{1,0} &= 0, B_{1,1} = 1, B_{1,2} = 5 \\
A_2 : B_{2,0} &= 0, B_{2,1} = 2, B_{2,2} = 7, B_{2,3} = 10 \\
A_3 : B_{3,0} &= 0, B_{3,1} = 3, B_{3,2} = 4
\end{aligned}$$

Die Tabellen $u(k, r)$ und $J(k, r)$ werden wie in Abbildung 3.18a aufgestellt. Die verfügbare Ressourcenbandbreite des Systems wurde für dieses Beispiel auf $R = 10$ normiert. Damit ergibt sich eine sehr grobe Ressourcenaufösung, für die Darstellung der Optimierung hier ist diese jedoch ausreichend. Der Tabellenindex k gibt die Anzahl der in Betracht gezogenen Anwendungen an. Index r stellt die verfügbare Ressourcenmenge (Bandbreite) für jeden Iterationsschritt dar. Für dieses Beispiel wurde außerdem mit $U_i(M_{i,j}) = j$ der Nutzen eines jeden Anwendungsmodus gleich seinem Index gesetzt. Alle Anwendungen sind außerdem mit $a_i = 1$ gleich priorisiert, die Ressourcen sollen also möglichst gleichmäßig verteilt werden.

$k \setminus r$	0	1	2	3	4	5	6	7	8	9	10
0											
1											
2											
3											

$k \setminus r$	0	1	2	3	4	5	6	7	8	9	10
1											
2											
3											

(a) Initialisierung von $u(k, r)$ und $J(k, r)$

$k \setminus r$	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	2	2	2	2	2	2
2											
3											

$A_1 : R_{1,0} = 0, R_{1,1} = 1, R_{1,2} = 5$
 $A_2 : R_{2,0} = 0, R_{2,1} = 2, R_{2,2} = 7, R_{2,3} = 10$
 $A_3 : R_{3,0} = 0, R_{3,1} = 3, R_{3,2} = 4$
 $U_i(M_{i,j}) = j$

(b) Iteration über r

$k \setminus r$	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	2	2	2	2	2	2
2	0	1	1	2	2	2	2	3	3	3	3
3	0	1	1	2	2	3	3	4	4	4	4

$A_1 : R_{1,0} = 0, R_{1,1} = 1, R_{1,2} = 5$
 $A_2 : R_{2,0} = 0, R_{2,1} = 2, R_{2,2} = 7, R_{2,3} = 10$
 $A_3 : R_{3,0} = 0, R_{3,1} = 3, R_{3,2} = 4$
 $U_i(M_{i,j}) = j$

(c) Iteration über alle Anwendungen

$k \setminus r$	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	2	2	2	2	2	2
2	0	1	1	2	2	2	2	3	3	3	3
3	0	1	1	2	2	3	3	4	4	4	4

$k \setminus r$	0	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	1	2	2	2	2	2	2
2	0	0	0	1	1	1	1	2	2	2	2
3	0	0	0	0	0	2	2	2	2	2	2

(d) Ergebnis mit beiden Tabellen

Abbildung 3.18: Beispiel für ein Optimierungsszenario

Die Optimierung beginnt in Abbildung 3.18b. Für $k = 0$ wird die oberste Reihe mit Nullen initialisiert. Die Berechnung einer jeden weiteren Reihe hängt von den Zahlen innerhalb der vorherigen Reihe ab. Der Algorithmus iteriert zunächst für Reihe $k = 1$ über die Ressourcen $r = 0..10$. Dabei wird in jedem Schritt der *höchste*¹¹ Modus von A_1 gesucht, dessen Ressourcenverbrauch in der jeweiligen Ressourcenmenge r hinein passt. Für $k = 1, r = 5$ zum Beispiel ist also Modus $M_{1,2}$ mit $B_{1,2} = 5$ der beste Kandidat. In $u(1, 5)$ wird dann der, durch die Selektion von $M_{1,2}$ errungene, Gesamtnutzen im System eingetragen. Innerhalb $J(1, 5)$ wird wie-

¹¹also der Modus mit dem größten Index und somit den höchsten Nutzen

derum der Index des ausgewählten Modus gespeichert. Da A_1 keinen höheren Modus als $M_{1,2}$ besitzt, verbleibt sein Nutzen innerhalb jeder weiteren Zelle bis $u(1,10)$. Die Berechnung der Reihe $k = 2$ bezieht sich nun bei jedem Schritt über r auf die Zahlen innerhalb $k = 1$ (siehe Abbildung 3.18c). Für $u(2,1)$ kann noch kein Modus von A_2 eingebracht werden, da der niedrigste eine Bandbreite von $B_{2,1} = 2$ hat. Für $u(2,3)$ ist jedoch $B_{2,1} = 2$ möglich, also wird mit $u(k-1, r-B_{k,j}) + a_k \cdot U_k(M_{k,j}) = u(2-1, 3-2) + 1 \cdot 1 = 2$ der neue erzeugbare Gesamtnutzen bestimmt. Die Überprüfung, ob $r - B_{k,j}$ immernoch innerhalb der Tabellengrenzen liegt, bestimmt die Gültigkeit einer partiellen Lösung. Somit liegt zum Beispiel $u(2,3)$ für $B_{2,2}$ mit $u(2-1, 3-7)$ außerhalb der Tabelle und ist keine gültige Lösung. Nachdem für $u(2,3)$ alle Modi von A_2 überprüft wurden, wird eine optimale Lösung mit dem höchsten erzeugbaren Nutzen ausgewählt – hier also der Modus $M_{2,1}$. Der Index 1 des ausgewählten Modus wird in $J(2,3)$ gespeichert. Bei $u(2,7)$ existieren zum Beispiel zwei Konfigurationen, die $r = 7$ genau erfüllen: $(B_{1,0}, B_{2,2})$ und $(B_{1,2}, B_{2,1})$. Letztere generiert jedoch einen höheren Nutzen von $u(2,7) = 3$ also wird diese ausgewählt und der Index 2 in $J(2,7)$ gespeichert. Die Berechnung wird auf diese Weise bis $u(3,10)$ durchgeführt. Es ist klar, in $u(3,10)$ befindet sich der höchste erzeugbare Nutzen mit der optimalen Konfiguration aller Anwendungen. Mit $J(k-1, r-R_{k,j})$ kann nun die Tabelle mit den selektierten Modi von ihrer letzten Zelle aus zurück-traversiert werden, um die optimale Konfiguration herauszufinden (siehe Abbildung 3.18d). Für A_3 wurde der also Modus $J(3,10) = 2$ ausgewählt. Mit $J(k-1, r-R_{k,j}) = (3-1, 10-4) = J(2,6) = 1$ ist für Anwendung A_2 Modus $M_{2,1}$ ausgewählt. Auf die gleiche Weise wird nun von $J(2,6)$ aus mit $J(2-1, 6-2)$ der Modus $M_{1,1}$ von A_1 selektiert. Die optimale Konfiguration für dieses Beispiel lautet also:

$$J(k, r) = \{M_{1,j_1}, M_{2,j_2}, \dots, M_{k,j_k}\} = \{M_{1,1}, M_{2,1}, M_{3,2}\}$$

mit einer totalen Ressourcenauslastung von $\sum B_{i,j} = 7/10$.

Priorität und Nutzen:

In Zeile 9 des auf Seite 109 präsentierten Algorithmus 1 wird ersichtlich, wie Anwendungsprioritäten innerhalb einer Lösung einbezogen werden, um Benutzerpräferenzen zu realisieren. Sie haben eine direkte Auswirkung auf den Nutzen $U_k(M_{k,j})$ der Anwendungsmodi. Damit alle Anwendungen gleichermaßen behandelt werden, müssen die Werte der Utility-Funktionen normiert werden. Dabei müssen $U_i, i \in \{1, 2, \dots, n\}$ auf denselben Wertebereich abbilden, um miteinander vergleichbar zu sein. Es ist außerdem von Vorteil, dass die Utility-Funktionen schnell berechnet werden können, da sie für jeden Wert $u(k, r)$ mehrmals aufgerufen werden. Komplex zu berechnende Utility-Funktionen können sich negativ auf die Ausführungszeit des Lösungsalgorithmus auswirken. Da im vorgestellten Systemmodell die Anzahl der Modi pro Anwendung diskret ist, ist es von Vorteil ihren Nutzen im Voraus zu berechnen und somit Verzögerungen zur Laufzeit zu vermeiden.

In den Experimenten in Kapitel 5.6 wurden vorberechnete Nutzen-Werte für die Applikationsmodi verwendet, die als Prozentsatz an Qualitätsgewinn eines jeden Modus in Bezug auf den höchsten Modus einer Anwendung definiert wurden (vgl. Kapitel 3.1, Gleichung 3.3). Diese Werte werden zur Laufzeit gemeinsam an den Optimierungsalgorithmus übergeben. Auf diese Weise sind keine weiteren Aufrufe von $U_i(M_{i,j})$ zur Laufzeit notwendig.

Theoretische Betrachtung:

Da das Rucksackproblem bekanntermaßen NP-vollständig ist [106], kann die Ausführungszeit von einem Algorithmus, der eine exakte Lösung bereitstellt, nicht polynomiell beschränkt werden. Mit dynamischer Programmierung kann jedoch eine Lösung mit pseudo-polynomieller Komplexität bereitgestellt werden. Nachdem die Anzahl der in $u(k, r)$ berechneten Werte $n * (R + 1)$ ist und für jeden Tabelleneintrag höchstens $m_{max} = \max_{i \in \{1, 2, \dots, n\}} \{|\mathcal{M}_i|\}$ Modi existieren, läuft der vorgestellte Algorithmus innerhalb einer pseudo-polynomieller Zeit, wenn die involvierten numerischen Werte innerhalb angemessener Wertebereiche gehalten werden.

Für die Anzahl der Anwendungen existiert theoretisch keine obere Grenze, für ein reales System sind jedoch mehr als 100 konkurrierende aktive Anwendungen eher unrealistisch. Ebenso wird angenommen, dass für keine reale Anwendung mehr als 5 Modi sinnvoll sein können. Des Weiteren kann der Wert der totalen Systemressourcen R auf einen beliebigen positiven Integer-Wert gesetzt werden. Je größer dieser Wert ist, desto präziser ist die Auflösung, mit der die Ressourcenverbräuche (Bandbreiten) der Anwendungen erfasst werden. Offensichtlich ist es jedoch auch von Vorteil für die Berechnungsdauer, den Wert von R relativ klein zu wählen, da in jedem Iterationsschritt über k innerhalb der Tabelle $u(k, r)$ auch über alle diskreten Werte $r \in \{0, 1, \dots, R\}$ der verfügbaren Ressourcenmenge R iteriert wird. Der Anstieg der Berechnungszeit für unterschiedlich große Werte für R wird in Kapitel 5.6 evaluiert.

Nachdem für die Lösung des Optimierungsproblems Integer-Programmierung verwendet wird, können einige partielle Optima gefunden werden, während der Algorithmus den Lösungsraum traversiert. Die Berechnung des Maximums in den Zeilen 8 bis 15 des auf Seite 109 vorgestellten Pseudo-Codes (siehe Algorithmus 1), bewirkt jedoch eine Bevorzugung von Applikationen mit kleinerem Index durch die Optimierung, wenn mehrere optimale Lösungen für ein Zwischenschritt existieren. Tauscht man das “>” auf Zeile 11 mit einem “ $\geq$ ” aus, führt das wiederum zu einer Bevorzugung von Anwendungen mit einem höheren Index. Innerhalb des vorgestellten Systemmodells würde dies bedeuten, dass Anwendungen aufgrund ihrer programmatischen Anordnung bevorzugt bzw. benachteiligt werden. Dies ist jedoch nicht akzeptabel, da die Ressourcen zwischen allen Anwendungen in einer konformen Art und Weise verteilt werden sollen. Dabei sollen allein die Benutzerprioritäten eine Auswirkung auf Präferenzen zwischen den Anwendungen haben. Um diese Charakteristik zu umgehen, kann der Algorithmus 1 leicht so erweitert

werden, dass er alle optimalen Zwischenlösungen nachverfolgt und z.B. eine “mittlere” Lösung als Optimum auswählt. Die erweiterte Lösung ist in Algorithmus 2 exemplarisch dargestellt.

Algorithm 2 ERWEITERTER OPTIMIERUNGSLGORITHMUS

```
1: ...
2: for  $k = 1 \rightarrow n$  do
3:    $u[k][0] \leftarrow 0$ 
4:   for  $r = 1 \rightarrow R$  do
5:      $max \leftarrow 0$ 
6:      $nmax \leftarrow 0$ 
7:     for  $M_{k,j} \in \mathcal{M}_k$  do
8:        $b \leftarrow u[k-1][r - R_{k,j}] + a_k \cdot U_k(M_{k,j})$ 
9:       if  $b > max$  then
10:         $max \leftarrow b$ 
11:         $J[k][r] \leftarrow J[k-1][r - R_{k,j}] \cup \{M_{k,j}\}$ 
12:      else
13:        if  $b = max$  then
14:           $maxima[nmax] \leftarrow M_{k,i}$ 
15:           $nmax \leftarrow nmax + 1$ 
16:        end if
17:      end if
18:    end for
19:    if  $nmax > 0$  then
20:       $J[k][r] \leftarrow J[k-1][r - R_{k,j}] \cup maxima[nmax/2]$ 
21:    end if
22:     $u[k][r] \leftarrow max$ 
23:  end for
24: end for
25: ...
```

Wie in den Zeilen 13 bis 16 zu sehen ist, werden multiple optimale Lösungen temporär in einem Vektor `maxima` zwischengespeichert. Die Auswahl eines mittleren Optimums wird auf Zeile 20 getroffen. Die Selektion eines „mittleren“ Pfads durch den Lösungsraum sorgt für eine gleichmäßigere Verteilung der verfügbaren Ressourcen zwischen den Anwendungen. Die Folge dessen ist eine potentiell *höhere Admissionsrate* von Anwendungen, was gegenüber einer ungleichen Verteilung mit nur wenigen, aber dafür mit höherer Qualität laufenden Applikationen, innerhalb eines ARTOS-Systems bevorzugt wurde.

3.2.4 Globale Betrachtung des Scheduling-Modells

In Abbildung 3.19 ist die Funktionsweise des Scheduling-Modells auf allen Ebenen im Überblick dargestellt. Hier wird von einer bereits etablierten und stabilen Konfiguration der Anwendungsmodi ausgegangen, bei der alle Tasks innerhalb ihrer Sicherheitspuffer (SB) approximiert sind. Solange alle Tasks innerhalb ihrer Toleranzgrenzen verbleiben, bleibt auch das System planbar und es finden keine Rekonfigurationen aufgrund ihrer realen Verbräuche statt. Dabei werden lediglich die mittleren Task-Kosten berechnet und periodisch bzw. instantan gegen ihre Approximationsgrenzen geprüft (siehe Kapitel 3.2.1).

Abbildung 3.19: Überblick über das Scheduling-Modell

Wenn eine Task aus ihrem SB ausbricht (1), entsteht zunächst eine neue Kostenschätzung $C_{i,k,l}$ (2), welche dann an die Anwendungsebene weitergereicht wird. Auf der Anwendungsebene wird dann eine lokale Planbarkeitsanalyse (3) für die betroffene Anwendung gemacht, um ihren veränderten Rechenzeitbedarf $R_{i,j}$ zu ermitteln (siehe Kapitel 3.2.2). Dabei entsteht auch eine neue Bandbreiten-Schätzung $B_{i,j}$ für den jeweiligen CBS der Anwendung. Aus diesem Grund muss nun auf der System-Ebene die Planbarkeit (4) der aktuellen Konfiguration der Anwendungsmodi überprüft werden (siehe Kapitel 3.2.3). Wenn diese eine höhere Bandbreitenmenge belegt als im System verfügbar ist oder wenn genügend Ressourcen für potentiell höhere Modi der Anwendun-

gen verfügbar sind, dann berechnet die Optimierung anhand der ermittelten Bandbreiten eine neue optimale Konfiguration. Wenn die Rekonfigurationsbedingungen nicht erfüllt sind, dann wird keine Optimierung durchgeführt und das System bleibt innerhalb seiner aktuellen (optimalen) Konfiguration planbar. Wurde jedoch eine neue Konfiguration berechnet, dann werden die Anwendungen in ihren neuen Zielmodi umgeschaltet (5). Dabei werden auch ihre CBS mit den entsprechenden Ressourcenkapazitäten und Perioden der Zielmodi neu konfiguriert. Ist eine Anwendung bereits in ihrem Ziel-Modus, findet keine Rekonfiguration der Anwendung statt.

Die Konsequenz aus diesem Szenario ist, dass der Burst einer einzigen Task zu einer Rekonfiguration des gesamten Systems führen kann. Entscheidend dafür wie oft solche Rekonfigurationen in einem System stattfinden, ist also die Güte, mit welcher Tasks mit stark schwankenden Ressourcenanforderungen durch das Modell approximiert werden. Abhängig davon ist theoretisch auch der Fall möglich, dass eine Task durch ein zyklisch schwankendes Verhalten periodisch Rekonfigurationen des Systems hervorruft. Die generelle Frage, die sich hier ergibt, ist, ob eine solche periodische Schwankung durch das Modell erkannt und erfasst werden kann, sodass die periodischen Rekonfigurationen des Systems unterdrückt werden können? Wie müssen die Ressourcen dabei am sinnvollsten verteilt werden? In Kapitel 5.5 wird anhand einer Fallstudie die Adaption der Sicherheitspuffer bei periodischen Schwankungen der Task-Verbräuche untersucht. Dort wird auch genau gezeigt, welche Schwankungen erkannt werden können und welche Maßnahmen für die optimale Verteilung der Ressourcen durch das Modell automatisch getroffen werden.

3.3 Zusammenfassung der Beiträge dieses Kapitels

In diesem Kapitel wurde ein Applikationsmodell für Anwendungen und Tasks mit mehreren Betriebsmodi und ein hierarchisches Scheduling-Modell für sie entwickelt. Das Applikationsmodell ist für Schwankungen innerhalb der Ressourcenanforderungen der Anwendungen ausgelegt und bietet Methoden an, mit denen die Kosten ihrer Tasks zur Laufzeit approximiert werden können. Die somit erzeugten Kostenschätzungen werden von einem hierarchischen Scheduler verwendet, welcher für die Steuerung der Anwendungen und ihrer Tasks entwickelt wurde. Eine Optimierung entscheidet wiederum über die Qualitäten und die Betriebsmodi der Anwendungen und befolgt dabei Benutzer-Prioritäten.

Einer der Hauptbeiträge dieses Kapitels ist der Entwurf eines erweiterten Constant Bandwidth Server-Mechanismus, welcher für die Steuerung ganzer Anwendungen mit mehreren Tasks und für eine Durchsetzung ihrer approximierten Ressourcenverbräuche ausgelegt wurde. Der Fokus

wurde hierbei besonders auf die Dimensionierung der Server gelegt, die eine Einhaltung der anwendungs-lokalen Task-Fristen garantiert. Andere Arbeiten im Umfeld von CBS beschäftigen sich z.B. mit der Lösung des *Deadline-Aging* Problems [107] bei CBS, oder definieren eine optimale Dimensionierung der Server mit der minimalen Anzahl ihrer gegenseitigen Verdrängungen [108]. In dieser Arbeit wird die Server-Dimensionierung auf eine optimale Abdeckung der kumulativen Rechenzeitanforderung der Anwendungs-Tasks mit genügend Rechenkapazität für eine minimale Periode des Servers ausgelegt. Letztere sorgt automatisch für genügend Verdrängungen der Server, so dass ihre internen Tasks fristgerecht aktiviert und fertig werden können. Somit realisiert das vorgeschlagene Modell ein CBS-internes Task-Scheduling (das in dieser Arbeit mit EDF implementiert wurde). Die dynamische Anpassung der Server-Kapazitäten an die realen Verbräuche der Anwendungen sorgt theoretisch auch für weniger Kostenüberläufe und bietet möglicherweise damit eine Lösung des *Deadline Aging*-Problems der CBS. Dies wurde jedoch im Rahmen dieser Arbeit nicht näher untersucht.

Ein weiterer wichtiger Beitrag dieses Kapitels ist das Approximations-Modell der Tasks, mit dem ihre Kostenschätzungen nur bei einer starken Abweichung vom real gemessenen Verhalten aktualisiert werden. Dies ermöglicht eine Einstellung des Systems in einen stabilen Zustand, bei dem Planbarkeitstests für die Anwendungen und die Optimierung ihrer Modi-Konfiguration nur unter bestimmten Bedingungen betrieben werden müssen. Dies spart Aufwand im System und verteilt die damit verbundene Komplexität über die Zeit.

Im nächsten Kapitel 4 wird nun das vorgeschlagene Applikations- und Scheduling-Modell mit Echtzeit-Java entwickelt und innerhalb einer konkreten Plattform-Konfiguration integriert. Auf diese Weise entsteht ein Prototyp des ARTOS-Systems, welcher im Kapitel 5 für die Evaluation der Scheduling-Mechanismen im hierarchischen Modell anhand realer Anwendungsszenarien verwendet wird.

4

Implementierung des Modells

Im Folgenden wird nach einer Beschreibung des gewählten Grundsystems eine Implementierung des vorgeschlagenen hierarchischen Scheduling-Modells von ARTOS mit Echtzeit-Java (Real-time Specification for Java (RTSJ)) beschrieben. Das Grundsystem wird zunächst mit einer Top-Down-Strategie von der Realisierung komponenten-basierter und dynamisch installierbarer Anwendungen mit OSGi, über die benötigten RTSJ-Mechanismen innerhalb einer Echtzeit-JVM bis hin zur Abbildung von Echtzeit-Aktivitätsträgern auf einer geeigneten Scheduling-Strategie innerhalb des unterliegenden Betriebssystem-Kernels vorgestellt. Die Implementierung des Modells mit RTSJ-Funktionalitäten und Datentypen wird später auf die einzelnen Ebenen des Modells heruntergebrochen und bottom-up vorgestellt. Dabei werden sukzessive neue Mechanismen hinzugefügt oder bereits bestehende erweitert. Bei jedem Schritt wird die Kompatibilität der Implementierung mit RTSJ bewertet und diskutiert. Da RTSJ nicht alle benötigten Vorgänge mit standardisierten Mechanismen unterstützt, sind auf mehreren Ebenen spezielle Anpassungen und Erweiterungen der RTSJ API notwendig, welche im Einzelnen vorgestellt werden. Zum Schluss dieses Kapitels wird nochmal kritisch auf Probleme mit RTSJ eingegangen, deren Lösungen Anpassungen grundlegender RTSJ-Mechanismen abverlangt haben. Da zum Zeitpunkt der Fertigstellung dieser Promotionsarbeit eine neue Version der Echtzeit-Java Spezifikation bearbeitet wurde, wurden die RTSJ-Erweiterungen zur Standardisierung vorgeschlagen und publiziert.

Auf der Basis der hier bereitgestellten Implementierung, wurde das vorgeschlagene Scheduling-Modell evaluiert, was später im Kapitel 5 näher vorgestellt wird.

4.1 Komponentenmodell und OSGi

OSGi [87] ist eine offene und standardisierte Dienstplattform auf der Basis von Java. Es unterstützt Komponenten (Bundles) und deren Lebenszyklus. Bundles können Bibliotheken, Anwendungen oder Dienste enthalten. Letztere werden über eine *Dienstregistratur* (*Service Registry*) anderen Bundles zugänglich gemacht. OSGi stellt also eine dienstbasierte Architektur (*Service Oriented Architecture (SOA)*) bereit, bei der Applikationen dynamisch aus unterschiedlichen Komponenten und Diensten zusammengesetzt werden können.

Abbildung 4.1: OSGi-Architekturmodell [87]

In Abbildung 4.1 ist das Architekturmodell eines OSGi Systems dargestellt. Ein OSGi-Framework wird innerhalb einer Standard Java Virtual Machine ausgeführt, welche wiederum auf einem nativen Betriebssystem aufsetzt. Über eine Abstraktion der darunterliegenden Ausführungsumgebung (*Execution Environment*) werden auf der *Modul-Ebene* die Bundles voneinander isoliert und Abhängigkeiten zwischen ihnen geregelt. Dafür definiert OSGi eine Hierarchie von Java *Class Loadern*¹, bei der jedes Bundle innerhalb einer eigenen Class Loader-Instanz gekapselt wird.

Abhängigkeiten zwischen den Bundles werden auf der Modul-Ebene anhand von *importierten* und *exportierten* Java-Paketen deklariert und spätestens beim Start der Komponenten aufgelöst. Sie werden durch ein Delegationsmodell zwischen den Bundle-Class Loadern umgesetzt, wobei

¹java.lang.ClassLoader Instanzen

das Framework beim Laden von Java Klassen die Interaktionen zwischen den Class Loadern der beteiligten Komponenten steuert und für Typkonsistenz sorgt. Solange sich die Konfiguration der Bundles innerhalb einer Plattform nicht ändert, bleibt das Delegationsmodell zwischen den Komponenten statisch.

Auf der höheren *Life-Cycle-Ebene* wird der Lebenszyklus der Bundles verwaltet; sie können dynamisch installiert, deinstalliert, gestartet und gestoppt werden, ohne einen Neustart des gesamten Frameworks abzuverlangen. Veränderungen der Abhängigkeiten zwischen den Komponenten werden dabei durch das Framework automatisch auf der Modul-Ebene geregelt.

Da Code-Abhängigkeiten zwischen den Komponenten auf der Modul-Ebene zu grobgranular und statisch sind, stellt OSGi auf der *Service-Ebene* ein Modell zum dynamischen Bereitstellen und Ausführen von Diensten bereit. Dabei können Bundles eigene Dienste über eine Registratur exportieren und dynamisch Dienste anderer Komponenten aufsuchen und aufrufen. Dieses Modell unterstützt die Trennung zwischen der Spezifikation und mehreren Implementierungen eines Diensts. Die Suche einer geeigneten Spezialisierung wird dabei durch ein LDAP-basiertes Beschreibungsprotokoll für nicht-funktionale Eigenschaften der Dienst-Implementierungen unterstützt.

Durch seine dynamisch ladbaren und ausführbaren Komponenten ist OSGi eine ideale Plattform für die Umsetzung des vorgestellten Scheduling-Modells. Anhand dessen kann eine variable Menge von Anwendungen in einem laufenden System relativ einfach realisiert werden. Zur Differenzierung unterschiedlicher Anwendungen wurde die Standard OSGi-Spezifikation zunächst mit einem Applikationsmodell erweitert, bei dem eine Anwendung durch mehrere Komponenten und ihren internen Aktivitäten definiert wird. Die Aktivitäten werden somit logisch zusammengeschlossen und im Sinne ihrer Ressourcenanforderungen abstrahiert.

Unterschiedliche Modi der Anwendungen und ihrer Aktivitäten können mit OSGi flexibel auf der Basis der dynamischen Dienst-Auswahl auf der Service-Ebene implementiert werden. Dabei kann eine Anwendungsaktivität den zuständigen Code für einen Modus dynamisch innerhalb der Dienstregistratur qualifizieren und über eine definierte Schnittstelle aufrufen. In OSGi4C [109] wurde bereits für ein Szenario gezeigt, wie eine dynamische Auswahl einer geeigneten Webserver-Komponente anhand von nicht-funktionalen Eigenschaften wie Speicherverbrauch oder Durchsatz realisiert werden kann. Ein ähnlicher Mechanismus kann auch auf der Dienstebene für die Komposition von Echtzeit-Anwendungen mit unterschiedlichen Betriebsmodi und Qualitäten etabliert werden.

Darüberhinaus wurde im Rahmen einer Vorarbeit *Recoverable Class Loaders (RCL)* [110] eine Lösung zur Beschleunigung des Starts komplexer Java-Programme durch das Speichern und

erneute Laden kompletter Objektstrukturen einer JVM entwickelt. Dazu gehört auch der Zustand der geladenen Klassen innerhalb wiederherstellbarer Class Loader-Instanzen. Da das Class-Loading generell ein unvorhersehbar-komplexer Engpass beim Start einer jeden Java-Anwendung darstellt [111], wurden die RCL-Mechanismen auch für OSGi-basierte Anwendungen integriert. Damit kann der Zustand eines Bundles sehr effizient wiederhergestellt und der Start einer ganzen OSGi-Plattform beschleunigt werden. Bei der Evaluation der Mechanismen auf verschiedenen mobilen Geräten zeigte sich ein Performanzgewinn von bis zu 96%.

Obwohl OSGi mittlerweile im Bereich ressourcenbegrenzter mobiler Geräte und eingebetteter Systeme weit verbreitet ist [112], unterstützt es noch kein dynamisches Management von Ressourcen oder die Einhaltung von Echtzeit-Anforderungen. Für eine Integration des vorgestellten Ressourcenmodells innerhalb OSGi sind jedoch periodische Aktivitäten mit Fristen notwendig, die in einer solchen Form durch Standard Java nicht unterstützt werden. Solche Mechanismen werden jedoch durch die Echtzeit-Spezifikation für Java bereitgestellt, welche im Folgenden vorgestellt wird.

4.2 Real-Time Specification for Java (RTSJ)

RTSJ [27] ist eine Java-Erweiterung, welche die Unvorhersagbarkeit der Ausführungszeit von Java-Code beseitigt und Konzepte für Echtzeitsysteme auf der Sprachebene einführt, ohne die Syntax von Java zu modifizieren. Zu den grundlegendsten Erweiterungen gehören die Definitionen von Echtzeit-Aktivitätsträgern mit festen prioritäts-basierten Scheduling-Mechanismen, die Erweiterung mit speziellen (Scoped- und Immortal-Memory) Speicherbereichen, bei denen die klassische Garbage-Collection nicht benötigt wird und ein asynchrones Event-Framework für RTSJ-spezifische und benutzerdefinierte Ereignisse mit steuerbaren und planbaren Event-Handlern. Die meisten RTSJ-fähigen JVMs (im Folgenden RT-JVMs genannt) können auch als Laufzeitumgebung für Standard Java-Programme verwendet werden. Sie unterscheiden sich in ihrer Realisierung (z.B. native Compiler vs. interpretierende VMs) und in der Implementierung optionaler Mechanismen wie Garbage-Collection, Kostenüberwachung oder Prozess-Gruppen. Die aktuelle RTSJ Version 1.0.2 wird in Kürze durch eine neue Version 2.0 abgelöst, die zusätzliche Erweiterungen und Modifikationen enthält. Im Folgenden werden einige Mechanismen aus RTSJ 1.0.2 grob vorgestellt, die bei der Implementierung des Scheduling-Modells in dieser Arbeit zum Einsatz kommen.

4.2.1 Echtzeit-Aktivitäten und Scheduling

In der aktuellen RTSJ Spezifikation 1.0.2 werden Echtzeit-Aktivitätsträger grundsätzlich als so genannte **Schedulable** Entitäten abstrahiert, die einem **Scheduler** zugeordnet werden können. Die Spezifikation lehnt **Schedulable**-Objekte an klassische Echtzeit-Tasks an. Sie werden als *Echtzeit-Threads* (**RealtimeThreads**) oder *Asynchrone Event-Handler* (AEH) spezialisiert und können *periodisch* oder *aperiodisch* sein. Für die Definition ihrer Aktivierungseigenschaften unterstützt RTSJ sog. **ReleaseParameters** mit entsprechenden Subklassen **AperiodicParameters** und **PeriodicParameters**. Der Unterschied zwischen den beiden Typen besteht darin, dass bei letzteren eine relative Periode (**period**) in Millisekunden angegeben werden kann. Darüber hinaus können Echtzeit-Aktivitätsträger mit einem absoluten Startzeitpunkt (**start**), einer (relativen) Frist (**deadline**) und mit Angaben über ihre Ausführungskosten (**cost**) versehen werden. Letztere dienen einer optionalen Überwachung und Durchsetzung von Thread-Kosten, um eine Überallokation der CPU zu verhindern, falls die entsprechende RT-JVM diese unterstützt.

Wie bereits erwähnt werden durch das ARTOS-Modell zunächst nur periodische Aktivitäten unterstützt. Die Kosten aperiodischer Echtzeit-Threads müssen beschränkt und zeitlich verteilt werden, um sie in einem periodischen System planen zu können. Dies kann anhand von Server-Mechanismen, wie z.B. Deferrable Server [34], erreicht werden, die solche Aktivitäten periodisch mit Ressourcen bedienen. Im Rahmen dieser Arbeit wurde auch ein Constant Bandwidth Server entwickelt, der ebenfalls zur Kapselung einzelner aperiodischer Aktivitäten verwendet werden kann. Nichtsdestotrotz wurden auf der RTSJ-Ebene zunächst nur Anwendungsaktivitäten mit **PeriodicParameters** in Betracht gezogen, die gemeinsam von einem CBS eingekapselt und bedient werden können.

Ein periodischer **RealtimeThread** wird mit einer bestimmten Frequenz aktiviert. Der Startzeitpunkt **start** innerhalb seiner **ReleaseParameters** bestimmt seine Aktivierungs-Phase, **period** seine Periodenlänge. Zwischen seinen Aktivierungsinstanzen kann sich ein Thread selbst bis zum Beginn seiner nächsten Periode mit der Funktion **RealtimeThread.waitForNextPeriod()** suspendieren. Dabei blockiert der Thread an einer Sperre innerhalb der RT-JVM, die erst zu Beginn seiner nächsten Periode wieder freigegeben wird. Hat sich ein Thread bis zum Zeitpunkt seiner relativen Frist oder seiner nächsten periodischen Aktivierung noch nicht freiwillig suspendiert, dann wird eine Fristverletzung (*Deadline Miss*) gemeldet. Dabei wird von der RT-JVM automatisch ein asynchrones Ereignis (**AsyncEvent**) ausgelöst, welches durch einen benutzerdefinierten Handler (**AsynchronousEventHandler**) abgefangen und behandelt werden kann. RTSJ spezifiziert jedoch keine Mechanismen für die Behandlung von Fristverletzungen. Eine Degradierung der Anforderungen und der Berechnungskomplexität von Echtzeit-Aktivitätsträgern ist appli-

kationsspezifisch und muss in der Regel durch die jeweilige Anwendung vollzogen werden. Im ARTOS-Modell wird die Degradierung der Anforderungen durch die Umschaltung der Anwendungsmodi unterstützt und gesteuert, wobei im Falle eines Engpasses die Perioden, die Fristen und auch die Berechnungskomplexität der Tasks relaxiert werden können.

Die Aktivierung von Echtzeit-Threads innerhalb einer RT-JVM wird durch einen **Scheduler** gesteuert. Dieser ist zunächst in RTSJ abstrakt definiert und kann durch unterschiedliche Verfahren mit eigenen Aktivierungs-Bedingungen und Semantik realisiert werden. Die Spezifikation unterstützt dafür sog. **SchedulingParameter**, die die Ausführungsdringlichkeit (*Execution Eligibility*) eines Threads definieren und z.B. als Priorität, Kritizität oder Laxheit (*Laxity*) für den jeweiligen Scheduler spezialisiert werden können. Trotz dieser allgemeinen Abstraktion wurde in der Spezifikation bisher nur ein *prioritätsbasierter präemptiver Scheduler* **PriorityScheduler** definiert. Anhand spezieller **PriorityParameters** werden Threads innerhalb entsprechender Prioritäts-Warteschlangen des **PrioritySchedulers** eingeordnet und dort mit einer FIFO-Reihenfolge abgearbeitet. Dabei unterstützt RTSJ 28 feste Prioritäten. Wird zu einem Zeitpunkt ein Thread mit einer höheren Priorität aktiviert, dann wird ein mit einer niedrigeren Priorität gerade aktiver Thread verdrängt. Darüber hinaus können die Prioritäten von Echtzeit-Threads dynamisch verändert werden. Dabei findet eine Umordnung der Prozesse innerhalb der Prioritäts-Warteschlangen des Schedulers statt. Die Prioritäten sind somit ein verstellbares Maß für die Wahrscheinlichkeit, mit der ein Aktivitätsträger CPU-Ressourcen bekommt. Die Realisierung des RTSJ **PrioritySchedulers**, auf der Basis eines Scheduling-Verfahrens innerhalb des einer RT-JVM unterliegenden Betriebssystems, wird in den Kapiteln 4.3.2 und 4.3.3 weiter thematisiert.

4.2.2 Asynchrone Ereignisverarbeitung

Die Spezifikation unterstützt eine Reihe von Ereignissen (**AsyncEvent**), die unter bestimmten Bedingungen automatisch ausgelöst und durch benutzerdefinierte Handler (**AsynchronousEventHandler**) abgefangen und behandelt werden können. Dazu gehören zum Beispiel Fristverletzungen und Kostenüberläufe von Aktivitäten und Aktivitätsgruppen (Processing Groups). Die benutzerdefinierten Handler-Instanzen für diese Ereignisse werden innerhalb der **ReleaseParameters** der Aktivitäten registriert und somit der RT-JVM bekannt gegeben. Die Handler können selbst eigene **ReleaseParameters** und **PriorityParameters** enthalten, die ihre Aktivierungseigenschaften und Priorität innerhalb des RTSJ **PrioritySchedulers** bestimmen.

Ein benutzerdefiniertes Ereignis kann durch eine Instanz eines `AsyncEvents` und einen `AsynchronousEventHandler` (AEH) realisiert werden. Das Ereignis wird mit dem Handler durch `AsyncEvent.addHandler()` verknüpft, es kann jedoch auch mehrere Handler besitzen. Ein Ereignis wird durch `AsyncEvent.fire()` ausgelöst und somit die Ausführung seiner Handler getriggert. Dabei wird von der JVM für jeden registrierten Event-Handler automatisch ein Aktivitätsträger erzeugt, der die Methode `handleAsyncEvent()` seiner AEH-Instanz ausführt. Diese Aktivitätsträger werden dann den Prioritäten und Aktivierungs-Parametern ihrer AEHs entsprechend in den Prioritäts-Warteschlangen des `PrioritySchedulers` neben den `RealtimeThreads` eingereiht und ausgeführt.

Eine Planbarkeitsanalyse für Event-Handler ist auch möglich, wenn auch schwierig, da zumindest eine Auftretts-Rate oder Wahrscheinlichkeit für die Ereignisse bekannt sein muss. Manche RT-JVMs unterstützen darum eine dedizierte Ereignis-Verarbeitung innerhalb bestimmter vorinitialisierter Echtzeit-Threads mit speziellen Aktivierungs-Eigenschaften. Die Kosten, die durch Ereignis-Behandlungen entstehen, können aber auch durch einen entsprechenden Server beschränkt und somit konstant innerhalb eines periodischen Systems eingeplant werden. In Kapitel 4.4.7 wird beispielsweise gezeigt, wie die Kosten für die periodische Überprüfung der Task-Approximationen und der damit verbundenen potentiellen Pufferaktualisierungen innerhalb asynchroner Ereignis-Handler erfasst und im Scheduling-Modell eingeplant werden können.

4.2.3 Asynchronous Transfer of Control

RTSJ unterstützt einen speziellen Mechanismus zur kontrollierten Unterbrechung eines Echtzeit-Aktivitätsträgers, genannt *Asynchronous Transfer of Control* (ATC). Mit einer speziellen Schnittstelle `javax.realtime.Interruptible` können auf der Code-Ebene Abschnitte definiert werden, innerhalb welcher ein Echtzeit-Thread durch eine `AsynchronouslyInterruptedException` (AIE) unterbrochen und kontrolliert beendet werden kann. Der Vorteil an ATC ist, dass diese Exception durch eine benutzerdefinierte Entscheidung explizit im Programm-Code ausgelöst werden kann.

```
1 public interface Interruptible{
2
3     void interruptAction(AsynchronouslyInterruptedException exception);
4
5     void run(AsynchronouslyInterruptedException exception)
6         throws AsynchronouslyInterruptedException;
7 }
```

Listing 4.1: Das `Interruptible` Interface

In Listing 4.1 ist die Definition der `Interruptible`-Schnittstelle dargestellt. Die Methode `run(AIE)` definiert den unterbrechbaren Code einer Implementierung, während `interruptAction(AIE)` automatisch durch die RT-JVM aufgerufen wird, wenn eine AIE ausgelöst wird. Die Ausführung einer bestimmten `Interruptible`-Instanz wird im Kontext einer AIE mit `aie.doInterruptible(Interruptible)` initiiert. Dabei wechselt die Kontrolle automatisch innerhalb des `Interruptible`-Objekts und ruft seine `run(AIE)`-Methode auf. Wird zu einem späteren Zeitpunkt die Exception durch `AIE.fire()` ausgelöst, dann wird die Ausführung der `run()` Methode abgebrochen und die `interruptAction()`-Methode aufgerufen. Während der gesamten Ausführung eines `Interruptibles` gelten die Aktivierungs-Eigenschaften des entsprechenden Echtzeit-Threads, welcher `doInterruptible()` aufgerufen hat. Die `Interruptibles` und ATC eignen sich sehr gut für die Realisierung von dynamisch selektierbarem und unterbrechbarem Code, was später bei der Implementierung der Task-Modi (in Kapitel 4.4.2) genutzt wird.

4.2.4 Processing Groups

RTSJ spezifiziert die optionalen `ProcessingGroupParameters` (PGP) [27], die als Basis-Mechanismus für die Implementierung von Prozessgruppen (*Processing Groups*) und Servern dienen. Mit PGPs können die Kosten einer Gruppe von `RealtimeThreads` gemeinsam überwacht und beschränkt werden. Dabei referenzieren alle Threads der Gruppe innerhalb ihrer `ReleaseParameters` die selbe PGP-Instanz. Der Gruppe wird dann eine feste CPU-Kapazität zugewiesen, die von den Echtzeit-Threads verbraucht und durch die RT-JVM periodisch aufgefüllt wird. Dafür enthalten PGPs spezielle `start`-, `period`- und `cost`-Parameter, mit denen der Start-Zeitpunkt des Servers, seine Allokations-Periode und eine Kapazität an CPU-Zeit in Millisekunden angegeben werden können. Die Parameter können mit entsprechenden Setter-Methoden dynamisch modifiziert werden. Die Kapazität des Servers wird am Anfang einer jeden Periode vollständig auf den Wert von `Cost` aufgefüllt. Die RT-JVM ist zur Laufzeit dafür zuständig, die Kosten der Threads innerhalb jeder PGP-Periode zu überwachen (*Cost Monitoring*). Wenn die Threads innerhalb einer Allokations-Periode mehr CPU-Zeit verbrauchen als erlaubt, wird von der JVM automatisch ein registrierter Überlauf-Handler (*Overrun Handler*) aufgerufen. Wenn die JVM Kostendurchsetzung (*Cost Enforcement*) unterstützt, dann werden die Threads bei einem Überlauf bis zum Beginn der nächsten Allokations-Periode suspendiert. Da die Überwachung und Durchsetzung von Kosten mit einem Overhead innerhalb einer RT-JVM verbunden sind, werden sie, und somit auch die Processing Groups, nicht von jeder JVM implementiert. Das Verhalten der Processing Groups ist komfortabel für die Realisierung speziell von *Deferrable Servers* [34], die ähnliche Laufzeiteigenschaften wie die PGPs besitzen.

4.2.5 RTSJ als Basis für ein Echtzeit-Komponentenframework

Wellings et al. untersuchen in [113, 114, 115] Möglichkeiten temporale Isolation und Admissionskontrolle für Bundles in ein RTSJ-basiertes echtzeitfähiges OSGi-Framework zu integrieren. Das Framework soll die Aktivierung neuer Bundles verweigern, falls dadurch die Ressourcen für bestehende Komponenten nicht mehr ausreichen würden. Dabei nehmen die Autoren Bundles als abgeschlossene Aktivitätseinheiten an, die keine Abhängigkeiten untereinander aufweisen. Für die temporale Isolation schlagen die Autoren *Deferrable servers* vor, welche bundle-interne Aktivitätsträger kapseln und ihren Ressourcenverbrauch beschränken sollen. Anhand der allozierten Server-Kapazitäten wird beim Start und Update von Komponenten eine Schedulability-Analysis durch eine Kombination von Response-Time-Analysis (RTA) [116], Boolean-Schedulability-Analysis [117] und Response-Time-Upper-Bound (RUB) [118] Berechnungen durchgeführt (für weitere Informationen siehe [115]). Dies entscheidet darüber, ob neue Komponenten gestartet werden können oder andere gestoppt werden müssen.

Das Bemessen der Server-Kapazitäten wird online auf Basis vordefinierter Worst-Case-Anforderungen der Komponenten [119] und nur bei Admission vollführt. Die WCET-Werte der Bundle-Aktivitäten hängen jedoch von einer Reihe bis zur Laufzeit unbekannter Faktoren ab [115], wie z.B. die dynamische Ausführung von unbekannten Diensten, die Behandlung synchroner Ereignisse im OSGi Framework, oder die Beteiligung applikations-interner Threads an der Garbage Collection. Darum alloziert der Algorithmus in [113] Bundle-Kapazitäten pessimistisch. Dies schafft jedoch bei schwankenden Anforderungen nur begrenzt Abhilfe. Weiterhin sollen *Service Contracts* [115] helfen, das Problem der dynamischen Komposition besser zu bewältigen. Mittels Annotationen können Dienstbenutzer WCET-Anforderungen deklarieren und die Menge der qualifizierten Dienstimplementierungen einschränken. Damit sollen verlässlichere WCET-Kalkulationen applikations-interner Aktivitäten möglich sein. Dienstentwickler müssen die WCET-Werte für ihre Implementierungen im voraus bestimmen und bereitstellen. Diese sind natürlich plattformspezifisch. Außerdem, können Dienste durch andere Dienste dynamisch komponiert sein, was durch die Autoren hier nicht weiter adressiert wird. Ferner können Bundles auch auf der OSGi Modulebene Codestücke anderer unbekannter Bundles ausführen und interagieren ohne die Dienstschicht von OSGi zu verwenden². Dieser Effekt verstärkt sich, wenn Bundles nicht in sich abgeschlossen sind, sondern unbekannte Dienste im Framework oder vorher unbekannte Codestücke nutzen. Eine Beschränkung auf in sich abgeschlossene Bundles würde die Motivation von OSGi und seinem Komponentenmodell gänzlich zur Diskussion stellen. Im Gegensatz zu den Arbeiten von Wellings et al. benötigt ARTOS keine WCET-Abschätzungen,

²z.B. mittels Import und Export von Java-Packages

sondern kann anhand aktueller Messungen Ausführungszeiten schätzen und entsprechende Qualitätsanpassungen betreiben.

4.3 Ausführungs-Plattform

RTSJ und OSGi bilden die Basis für einen Prototyp, da sie ideal die Grundidee von dynamisch ladbaren Anwendungen mit Echtzeitanforderungen unterstützen, die auf unterschiedlichsten Hardware-Plattformen ausgeführt werden können. Dabei ist RTSJ, was die Implementierungsebene der spezifizierten Mechanismen betrifft, offen. Diese können zum Beispiel als System-on-Chip innerhalb einer spezialisierten Hardware oder auch innerhalb bzw. anstelle eines Betriebssystems integriert werden [11]. Die gängigste Realisierung ist jedoch innerhalb einer RT-JVM, die auf einem Echtzeit-Betriebssystem aufsetzt. Dabei hängt es von der tatsächlichen JVM-Implementierung ab, welche Hardware-Plattformen und Betriebssysteme unterstützt werden.

4.3.1 JamaicaVM

Für die Implementierung und die Evaluation des vorgestellten Modells wurde im Rahmen dieser Arbeit die *JamaicaVM* [120] der Firma Aicas verwendet³. Diese unterstützt grundsätzlich 32- und 64-Bit Architekturen wie x86, ARM und PPC. Eine Portierung der JVM existiert bereits für viele Betriebssysteme, unter anderem für Linux, Windows CE/Embedded, VxWorks, QNX, RTEMS, Solaris, ucLinux usw. In dieser Arbeit wurde eine Linux-Variante der JVM verwendet.

Die JamaicaVM besitzt noch keinen Just-In-Time (JIT) Compiler. Um die Laufzeit-Performanz von Java-Programmen zu verbessern, unterstützt die JVM eine vorzeitige (Ahead-of-Time (AOT)) Übersetzung von Anwendungen in nativen Code für eine bestimmte Plattform – die JVM dient also auch als Cross-Compiler. Dabei wird das Anwendungsprogramm als Binärprogramm für die gewählte Zieplattform kompiliert, das einen Teil der benötigten Java-Laufzeitumgebung nativ beinhaltet.

JamaicaVM ist eine der wenigen RT-JVMs, welche die in RTSJ optionale Kostenüberwachung (Cost Monitoring) für Echtzeit-Aktivitäten und die Processing Groups implementiert. Nachdem diese Mechanismen für die Implementierung der Server-Mechanismen auf der Applikations-Ebene des hierarchischen Scheduling-Modells benötigt werden, wurde sie als Basis für seine Implementierung bevorzugt. Außerdem arbeitet die Firma Aicas aktuell an einer neuen Version

³Andere JVMs wurden zwar auch in Betracht gezogen, erwiesen sich jedoch aufgrund fehlender Cost-Monitoring-Mechanismen als ungeeignet.

2.0 der RTSJ-Spezifikation und ist für die Referenz-Implementierung verantwortlich. Der enge Kontakt mit den Aicas-Entwicklern erleichterte das Verständnis über ihre Implementierung bestimmter RTSJ 1.0.2 Mechanismen und ihre Erweiterungen in RTSJ 2.0.

4.3.2 Abbildung von Echtzeit-Threads innerhalb des Betriebssystems

Die JamaicaVM bildet Echtzeit-Aktivitäten (*RealtimeThread*) als Prozesse innerhalb des nativen Betriebssystems ab. Im Fall von Linux werden POSIX Threads (**pthread**s) als Grundlage verwendet. Für **pthread**s definiert die POSIX Spezifikation 1003.1b (Part 1: System Application Program Interface) [121] drei Scheduling-Strategien **SCHED_OTHER**, **SCHED_RR** und **SCHED_FIFO**. Diese haben eine unterschiedliche Semantik und belegen auch unterschiedliche Prioritätsräume des Kernels. Sie haben somit unterschiedliche Implikationen auf das Laufzeitverhalten und Garantien der Prozesse.

SCHED_OTHER ist eine implementierungs-abhängige Strategie, die bei Linux für die Realisierung des *Completely Fair Schedulers* (CFS) [122] verwendet wird. Dieser Scheduler implementiert ein präemptives Zeitmultiplexing-Modell, bei dem jeder Prozess⁴ ein Zeitquantum an CPU-Zeit bekommt. Anhand von sog. *nice*-Werten kann die Größe dieser Zeitquanten dynamisch modifiziert werden. Dabei versucht der CFS eine „faire“ Proportion an Prozessorzeit zwischen allen Prozessen einzustellen, bei der aber auch die Interaktivität von I/O-lastigen Prozessen erhöht wird [122]. Das CFS-Scheduling bietet keine Echtzeit-Garantien für die bedienten Prozesse.

SCHED_FIFO implementiert eine First-In-First-Out Strategie mit festen Prioritäten, bei der Prozesse in Warteschlangen eingeordnet werden und nach ihrer Wartezeit abgearbeitet werden. Dabei wird der Prozess mit der längsten Wartezeit als nächstes ausgeführt. Die Warteschlangen sind nach Prioritäten organisiert, die POSIX Spezifikation schreibt mindestens 32 Prioritäten (also 32 FIFO-Listen) vor. Ein aktueller Linux-Kernel (wie z.B. der in dieser Arbeit verwendete Linux-Kernel 3.10.65-rt69) implementiert 100 FIFO-Prioritätsstufen. Die POSIX **SCHED_FIFO** Strategie kann nach [121] grob wie folgt zusammengefasst werden:

- Wenn ein Prozess mit einer bestimmten Priorität dem FIFO-Scheduler hinzugefügt (also aktiviert) wird (**sched_setscheduler()**) oder die Priorität eines laufenden Prozesses dynamisch verändert wird (**sched_setparam()**), dann wird dieser am Ende der entsprechenden Prioritäts-Liste eingefügt.

⁴In Linux ist jeder Thread ein separater Prozess mit speziellen Einstellungen unter anderem für den gemeinsam mit anderen Prozessen verwendeten Adressraum [122]

- Wenn ein Prozess ausgeführt wird, dann wird er aus seiner Liste entfernt und der CPU zugewiesen. Prozesse können nur am Anfang (Kopf) der Warteschlangen entfernt werden.
- Wenn ein laufender Prozess die Funktion `sched_yield()` aufruft, dann gibt er freiwillig die Kontrolle ab und wird am Ende seiner Prioritäts-Liste eingefügt.
- Wird ein Prozess mit einer höheren Priorität zur Ausführung bereit, dann wird ein gerade laufender Prozess verdrängt. Wenn ein laufender Prozess verdrängt wird, dann wird er am Anfang seiner Prioritäts-Liste zurückgesetzt.
- Wenn ein laufender Prozess blockiert, dann wird er, solange die Blockierung dauert, innerhalb einer speziellen Liste zwischengespeichert. Wenn ein blockierter Prozess wieder lauffähig wird, dann wird er am Ende seiner Prioritäts-Liste eingefügt.

Der Prioritätsraum von `SCHED_FIFO` ist über den von `SCHED_OTHER` angeordnet, wodurch Threads mit dieser Scheduling-Strategie generell eine höhere Ausführungswahrscheinlichkeit besitzen. Die Vergabe von CPU-Zeit an Prozesse mit `SCHED_FIFO` hängt von der Last der Prozesse mit höheren Prioritäten ab.

`SCHED_RR` ist wiederum eine Abwandlung von `SCHED_FIFO`, bei der Prozesse zusätzlich ein Ausführungs-Zeitquantum bekommen. Okkupiert ein Prozess die CPU länger als sein Zeitquantum vorgibt, so wird er am Ende seiner Prioritäts-Liste zurückgestellt. Auf diese Weise wird verhindert, dass ein Prozess die CPU für sich monopolisiert. `SCHED_RR` operiert im selben Prioritätsraum wie `SCHED_FIFO`.

Mit einer entsprechenden Planung über den FIFO-Scheduler des Betriebssystems, wie zum Beispiel durch ein *Rate Monotonic* (RM) Scheduling, können nun Echtzeit-Garantien für die Prozesse unterstützt werden. In dieser Arbeit wurde anhand von Verschiebungen von Prozessen zwischen unterschiedlichen Prioritäts-Listen, ein EDF-basierter Scheduling-Mechanismus für Echtzeit-Tasks und ganzer Anwendungen entwickelt, welcher später in den Kapiteln 4.4.3 und 4.4.4 näher vorgestellt wird.

4.3.3 Abbildung von Prioritäten und CPU-Affinitäten

Die JamaicaVM unterstützt generell eine Abbildung von Echtzeit-Aktivitätsträgern auf `pthread`s mit der `SCHED_FIFO` Scheduling-Strategie (und optional mit `SCHED_RR`). Dabei wird der FIFO-Scheduler des Betriebssystems durch den `RTSJ PriorityScheduler` innerhalb der JVM abstrahiert. Dieser implementiert somit eine *Fixed Priority Preemptive* (FPP) Strategie.

Für die Abbildung der 28 RTSJ-Prioritäten innerhalb des Prioritätsraums von `SCHED_FIFO`, stellt die JamaicaVM so genannte *Priority-Maps* bereit. Über einen JVM-Parameter

```
-DJAMAICAVM_PRIMAP=1..10=0, sync=1, 11..38=49..77/FIFO
```

können, wie in diesem Beispiel, einzelne oder ganze Bereiche von RTSJ-Prioritäten im Prioritätsraum von `SCHED_FIFO` abgebildet werden. Dabei unterstützt die JamaicaVM eine eigene Aufteilung des RTSJ-Prioritätsraums. Die ersten 10 Prioritäten sind für Nicht-Echtzeit-Threads⁵ reserviert, diese werden im Beispiel weiter oben auf einer einzigen FIFO-Priorität abgebildet. Mit der `sync`-Priorität realisiert die JVM dann ein eigenes „best-effort“ Scheduling für die nicht-echtzeit Threads [123]. Die 28 RT-Prioritäten für die `RealtimeThreads` werden im Beispiel weiter oben auf einen benutzerspezifischen Bereich (49 - 77) innerhalb des FIFO-Prioritätsraums abgebildet.

Für eine einwandfreie Funktion des Scheduling-Modells und eine präzise Evaluation, ist es von Vorteil die RTSJ-Prioritäten so abzubilden, dass Echtzeit-Tasks nicht mit anderen System- oder Anwendungsaktivitäten innerhalb der selben FIFO-Prioritäten konkurrieren oder durch diese unterbrochen werden. Dafür kann ein relativ hoch-priorer freier Bereich gewählt werden⁶. Da die `SCHED_FIFO` Strategie innerhalb des Linux-Kernels einen höheren Prioritätsraum als `SCHED_OTHER` belegt, werden die Echtzeit-Tasks innerhalb des unterliegenden Betriebssystems nicht durch unkritische Nicht-Echtzeit-Aktivitäten beeinflusst.

Darüber hinaus können in einer Multikern-Umgebung System-Aktivitäten und Dienste auf bestimmte Cores ausgelagert werden, um einen oder mehrere dedizierte Kerne für die RT-JVM zu allozieren. Unter Linux kann hierfür das `taskset`⁷ Werkzeug verwendet werden, mit dem CPU-Affinitäten der laufenden Linux-Prozesse inspiziert und explizit gesetzt werden können. Diese haben dann eine sofortige Auswirkung darauf, auf welchem Core der jeweilige Prozess ausgeführt wird. Die JamaicaVM unterstützt grundsätzlich eine *Single-Processor*- und eine *SMP*-Variante. Da das hierarchische Scheduling Modell von ARTOS nicht für mehrere Prozessoren ausgelegt ist, wird im Rahmen dieser Arbeit die Ein-Prozessor-Variante der JVM verwendet. Auf welcher Recheneinheit innerhalb eines Multi-Kern-Systems die von der JVM erzeugten Prozesse ausgeführt werden, kann ebenfalls durch Festlegung von CPU-Affinitäten beim Start der JVM bestimmt werden. Dafür unterstützt die JamaicaVM einen entsprechenden JVM-Parameter,

```
-Djamaica.java_thread_default_affinity=7,8,9
```

⁵also `java.lang.Thread` Instanzen

⁶Mit dem Linux-Tool `ps` kann die aktuelle Belegung des FIFO-Prioritätsraums (`RTPRIO`) inspiziert werden

⁷siehe <http://manpages.ubuntu.com/manpages/wily/man1/taskset.1.html>

mit dem eine Liste der zu verwendenden CPU-Kerne anhand ihrer IDs⁸ angegeben werden kann. Im Fall von der Single-Core Variante der JVM wird hier nur eine ID angegeben und alle JVM-internen Prozesse laufen ausschließlich auf den entsprechenden Kern.

4.3.4 Voll-präemptiver Linux-Kernel: CONFIG_PREEMPT_RT

Als Basis der Ausführungsplattform wurde ein mit dem CONFIG_PREEMPT_RT Patch [124] (weiter nur RT-Preempt genannt) angepasster Linux-Kernel verwendet. Dieser Patch konvertiert einen Linux-Kernel zu einem sog. *voll-präemptiven* Kernel, welcher mit speziellen Mechanismen für die Entwicklung und Unterstützung von Echtzeit-Anwendungen ausgestattet ist. Die JamaicaVM wiederum setzt mit ihrer Funktionalität auf diesen Kernel auf und verwendet einige der bereitgestellten Mechanismen für die Implementierung der RTSJ-Spezifikation.

Ein traditioneller Linux-Kernel erlaubt nur unter bestimmten Umständen, dass ein Prozess einen anderen Prozess verdrängt: a) wenn die CPU gerade User-Mode Code ausführt, b) wenn der Kernel infolge einer Interrupt- oder Systemcall-Verarbeitung wieder verlassen wird und c) wenn der Kernel-Code an einem *mutex* blockiert oder explizit die Kontrolle an einem anderen Prozess abgibt (*yield*). Wenn also, während ein Prozess mit einer niedrigeren Priorität Kernel-Code ausführt, ein Ereignis ausgelöst wird, welches die Aktivierung eines höher-priorisierten Threads verlangt, dann kann der höher priorisierte Prozess nicht den im Kernel laufenden nieder-priorisierten verdrängen, bis dieser explizit die Kontrolle abgibt oder den Kernel verlässt. Dies kann zu einer hohen Latenz von mehreren hundert Millisekunden für die Aktivierung des höher priorisierten Thread führen [124].

Seit der Version 2.6 unterstützt der Linux Kernel die Konfigurations-Option CONFIG_PREEMPT, mit der der gesamte Kernel-Code außerhalb von spinlock-geschützten Regionen und Interrupt-Handlern durch einen höher priorisierten Prozess im Kernel unfreiwillig verdrängt werden kann. Mit dieser Option fällt die Aktivierungslatenz von Prozessen auf einige wenige Millisekunden. Zusätzliche Latenzen können nun durch Interrupt-Handlern innerhalb Geräte-Treibern erzeugt werden. Wenn eine Echtzeit-Anwendung kleinere Latenzzeiten als wenige Millisekunden benötigt, kann der CONFIG_PREEMPT_RT Patch [124] eingesetzt werden.

Der RT-Preempt Patch macht den Linux-Kernel völlig verdrängbar. Dafür werden zunächst Interrupt-Handler in präemptive Kernel-Threads umgewandelt. Ein gepatchter Kernel behandelt Interrupts im Kontext von Kernel-Threads, die auf die gleiche Art wie User-Space-Prozesse mit einem `task_struct` Process Control Block (PCB) realisiert sind. Darüber hinaus impliziert

⁸Innerhalb eines Linux Systems können die verfügbaren Kerne und ihre IDs in `/proc/cpuinfo` eingesehen werden.

der RT-Preempt Patch Anpassungen der kernel-internen Lock-Primitiven. Typische `spinlocks` werden mit präemptiven `rtmutexes` ersetzt. Kritische Bereiche, die auf die `spinlock_t` bzw. `rwlock_t` API aufsetzen, sind nun implizit verdrängbar. Nicht-präemptive Bereiche im Kernel können durch eine API-Erweiterung mit speziellen `raw_spinlock_t` realisiert werden. Um gegen Probleme wie Prioritätsumkehrung vorzugehen, implementieren die neuen `spinlocks` im Kernel ein Prioritätsvererbungs-Protokoll (*Priority Inheritance*) [125]. Zu guter Letzt wird mit dem RT-Preempt Patch die Linux Timer-API für hoch-auflösende Kernel-Timer und Timeouts erweitert, womit im User-Space POSIX Timer mit einer hohen Auflösung bereitgestellt werden.

Die JamaicaVM setzt auf die durch den RT-Preempt bereitgestellten `spinlock` und `timer` Mechanismen für die Implementierung der RTSJ-internen Locking- und Timer-Mechanismen auf, wenn diese im Kernel verfügbar sind. Da die Kernel `spinlocks` nur das Priority Inheritance-Protokoll unterstützen, RTSJ aber auch *Priority Ceiling*-Mechanismen [125] abverlangt, implementiert die JamaicaVM letztere auf einer höheren Ebene selbst.

4.4 Implementierung mit RTSJ

Der hierarchische Scheduling-Ansatz wurde basierend auf Echtzeit-Java (RTSJ [27]) mit der JamaicaVM [120] sowie OSGi [87] implementiert und als Framework für unbekannte Echtzeit-Java-Anwendungen realisiert. Das Framework wurde ARTOS genannt [126]. Abbildung 4.2 veranschaulicht die Architektur der Lösung. Mit OSGi konnte eine dynamische Bereitstellung, Aktivierung und Deaktivierung von Anwendungen einfach realisiert werden. Dafür wurde die Standard OSGi API auf der Basis einer bereits existierenden Implementierung – Concierge OSGi [112] – mit entsprechenden Typ-Definitionen für multi-modale Anwendungen (**Application**) und Aktivitäten (**Task**) erweitert. Instanzen dieser Klassen werden automatisch in das hierarchische Scheduling-Modell eingebettet. Das Modell selbst wurde als Teil des OSGi Frameworks über RTSJ und der JamaicaVM integriert. Somit unterstützt ein ARTOS-Framework einerseits traditionelle OSGi-Bundles, aber auch sog. ARTOS-Anwendungen mit Echtzeitanforderungen. Beide Seiten, die Echtzeit- und die Nicht-Echtzeit-Komponenten, können untereinander Code und Dienste teilen und dynamisch in Anspruch nehmen. Dies kann zu Variationen des Code-Pfads von Echtzeit-Tasks und somit des Ausführungsverhaltens von Anwendungen führen, was durch das Scheduling-Modell automatisch erfasst und geregelt werden soll. In diesem Sinne implementiert die Kombination von ARTOS mit OSGi eine Lösung für eine OSGi-basierte, echtzeitfähige Middleware-Plattform. Der vollständige Code der Implementierung ist offen und befindet sich in [127].

Abbildung 4.2: Integration von ARTOS innerhalb von OSGi

4.4.1 Realisierung von Applikationen und Applikationsmodi

Zur Implementierung von multimodalen ARTOS-Anwendungen werden zunächst die zwei Basis-Typen **Application** und **ApplicationMode** eingeführt. Diese Klassen dienen als Container für die Definitionen der Applikations-Tasks und ihrer Modi und enthalten unter anderem Funktionen für den Start und den Stop von Applikationen sowie für den Wechsel ihrer Betriebsmodi. Anhand dieser Basis-Typen werden tatsächliche Anwendungs-Implementierungen in die bereits vorgestellten Scheduling-Mechanismen auf allen Ebenen des hierarchischen Modells integriert. Im Folgenden werden nur die Mechanismen innerhalb **Application** und **ApplicationMode** grob vorgestellt, welche für das Verständnis der weiteren Implementierung des Modells notwendig sind. Diese Definitionen werden im Verlauf des Kapitels bei Bedarf sukzessive erweitert. Die vollständige Implementierung dieser Klassen, wie auch der internen Vorgängen bei der Installation und der Aktivierung der Anwendungen, befindet sich in [127].

Die Definition einer Anwendung in Listing 4.2 enthält eine Liste ihrer Betriebsmodi **modes**, eine Referenz auf ihren aktuellen Modus **activeMode** und eine Anwendungspriorität. Die Referenzen auf den Task-Scheduler und den CBS der Anwendung werden für die Implementierung des Scheduling auf der Task- bzw. Applikations-Ebene benötigt, welches später in den Kapiteln 4.4.3 und 4.4.4 vorgestellt werden. Durch **switchMode()** wird ein bestimmter Betriebsmodus beim Start einer Anwendung oder später zur Laufzeit, aufgrund von Entscheidungen auf der System-Ebene des Modells, aktiviert.

```
1 public class Application {
2     Map<Integer, ApplicationMode> modes;
3     public ApplicationMode activeMode;
4     TaskSchedulerEDF taskScheduler;
5     int priority = 1;
6     CBS cbs;
7
8     public void switchMode(ApplicationMode newMode) {
9         activeMode = newMode;
10        activeMode.activate();
11    }
12    ...
13 }
```

Listing 4.2: Applikation

Der Betriebsmodus einer Anwendung wird in Listing 4.3 definiert. Er enthält zunächst einen vorausberechneten numerischen Nutzen `utility`, welcher für die Optimierung der Modi in Kapitel 4.4.5 benötigt wird, und eine Referenz auf seiner Applikation `app`. Da ein Modus eine bestimmte Konfiguration von Task-Modi impliziert (siehe Kapitel 3.1), wird in `taskModes` eine Liste der entsprechenden Zielmodi der Anwendungs-Tasks für den aktuellen Modus ihrer Anwendung verwaltet.

```
1 public class ApplicationMode {
2     int utility;
3     Application app;
4     Map<Task, TaskMode> taskModes;
5
6     public void activate() {
7         for (Task task : taskModes.keySet()) {
8             task.activateMode(taskModes.get(task));
9         }
10
11        app.cbs.reconfigure();
12    }
13 }
```

Listing 4.3: Applikationsmodus

In `activate()` werden die Vorgänge zur Aktivierung eines Anwendungsmodus definiert. Zunächst werden durch `task.activateMode()` alle Tasks in ihren entsprechenden Zielmodi überführt, was im Kapitel 4.4.2 weiter erläutert wird. Nachdem die Task-Modi umgeschaltet wurden, wird der CBS der Anwendung für ihren neuen Betriebsmodus rekonfiguriert. Dieser Vorgang wird

in Kapitel 4.4.4 weiter erklärt. Eine explizite Deaktivierungs-Funktion für einen Anwendungsmodus wird hier nicht benötigt. Wie später in Listing 4.5 gezeigt wird, wird bei der Umschaltung mit `Task.activateMode(TaskMode)` der alte Modus einer Task automatisch deaktiviert und ein neuer Modus aktiviert. Der Stopp einer Anwendung wird hier nicht näher behandelt, dieser ist jedoch im Code in [127] ersichtlich⁹.

4.4.2 Realisierung von Tasks und Task-Modi

Im Folgenden wird eine spezialisierte API auf der Basis von RTSJ präsentiert, welche das ARTOS-Scheduling-Modell unterstützen soll. Anhand dessen kann das applikations-lokale EDF-Scheduling, die Kostenerfassung der Tasks und auch eine dynamische Selektion und Umschaltung von Anwendungs- und Task-Modi realisiert werden. Bislang unterstützt das präsentierte Modell nur periodische Tasks, aperiodische Tasks können jedoch gleichermaßen bedient werden, z.B. durch Kapselung in aperiodischen Servern (z.B. Deferrable Servers [34]). Die Realisierung solcher Server mit RTSJ wurde generell durch Burns und Wellings in [128, 129] diskutiert. Eine Implementierung von Constant Bandwidth Servern mit RTSJ wird später innerhalb dieser Arbeit präsentiert.

Wie bereits in Kapitel 4.2.1 erklärt wurde, spezifiziert RTSJ für periodische Echtzeit-Aktivitäten `RealtimeThreads`, die mit speziellen Parametern vom Typ `PeriodicParameters` mit dynamisch verstellbaren Perioden, Kosten und Deadlines initialisiert werden. Wenn Kostenschätzungen für die Threads verfügbar sind, können diese von einer RT-JVM durchgesetzt werden. Innerhalb von ARTOS werden diese jedoch nicht verwendet, da das Framework die Kosten der `RealtimeThreads` selbst misst und anhand von Schätzungen ihres Verbrauchs entsprechende Anwendungskapazitäten mit Constant Bandwidth Servern auf der Applikations-Ebene durchsetzt. Die periodischen Aktivierungen der Echtzeit-Threads werden dann durch die RT-JVM und den prioritätsbasierten FIFO Scheduler `PriorityScheduler` von RTSJ gesteuert (vgl. Kapitel 4.3.2).

Bei der Implementierung von ARTOS wurden die Echtzeit-Threads von RTSJ nur als Hülle für die tatsächliche Logik der periodischen Tasks verwendet. Letztere muss bei Wechsel der Task-Modi dynamisch, im Sinne ihrer Implementierung und zeitlichen Anforderungen, austauschbar sein. Darum wurden in ARTOS Task-Modi als Instanzen von `javax.realtime.Interruptible` realisiert (vgl. Listing 4.4). Wie in Kapitel 4.2.3 bereits erklärt wurde, ist ein `Interruptible`¹⁰

⁹siehe `appStop()` in <https://github.com/vladnikolov/arema/blob/master/src/framework/src/arema/framework/application/Application.java>

¹⁰`Interruptibles` sind Teil eine RTSJ-spezifischen Mechanismus genannt “Asynchronous Transfer of Control” (ATC) [27] (vgl. Kapitel 4.2.3)

ein selektiv ausführbarer Code-Abschnitt, welcher zu einer beliebigen Zeit aufgerufen und, wenn nötig, abgebrochen werden kann. Jede `TaskMode`-Spezialisierung muss eine Interface-Methode `run()` überschreiben, welche die tatsächliche modus-spezifische Logik umgibt. Dies ermöglicht die Implementierung unterschiedlicher Modi und ihre Umschaltung, wobei für eine Task dynamisch eine andere `Interruptible`-Instanz, also ein unterschiedlicher `TaskMode`, ausgewählt und ausgeführt werden kann. Bei Modi-Wechsel werden die Methoden `activate()` und `deactivate()` automatisch durch das Framework aufgerufen. Sie ermöglichen modus-abhängige Anpassungen der Tasks, wie zum Beispiel die Konvertierung benötigter Berechnungsdaten oder die Synchronisation mit einer anderen Datenquelle. Mit `resSample()` werden zu guter Letzt die für den jeweiligen Modus gemessenen Task-Kosten pro Periode und ihr gleitendes Mittel berechnet und gespeichert. Eine Implementierung dieser Methode wird in Kapitel 4.4.7 bereitgestellt und zusammen mit der Approximation der Task-Kosten behandelt.

```
1 public abstract class TaskMode implements Interruptible {
2     abstract void activate();
3     abstract void deactivate();
4     PeriodicParameters release;
5
6     /* the actual task-mode logic */
7     abstract void
8     run(AsynchronouslyInterruptedException exception)
9         throws AsynchronouslyInterruptedException;
10
11     /* compute moving average of measured costs */
12     void resSample(double measuredValue) {...}
13
14     RelativeTime getPeriod() {
15         return release.getPeriod();
16     }
17
18     RelativeTime getDeadline() {
19         return release.getDeadline();
20     }
21 }
```

Listing 4.4: Task-Modus

Jede Task besitzt eine vordefinierte Instanz eines speziellen Task-Modus `TaskZeroMode`, welcher für die Implementierung der Applikationsmodi $M_{i,0}$ (siehe Kapitel 3.1) benötigt wird. Wenn der Null-Modus einer Task aktiviert wird, wird der entsprechende Echtzeit-Thread vom `PriorityScheduler` entfernt (vgl. `RealtimeThread.deschedulePeriodic()` in RTSJ [27]). Dabei wird der Thread kontrolliert am Ende seiner aktuellen Periode suspendiert und nicht mehr periodisch aktiviert. Wenn zu einem späteren Zeitpunkt der Null-Modus eines Threads deaktiviert wird, dann wird dieser wieder beim `PriorityScheduler` registriert (vgl.

`RealtimeThread.schedulePeriodic()` in RTSJ [27]) und periodisch ausgeführt. Diese Prozedur hat inhärente Auswirkungen auf das Scheduling auf der Task-Ebene des ARTOS-Modells, welche später in Kapitel 4.5.1 erklärt werden. Die entsprechende Definition einer (periodischen) Task und die Prozedur der dynamischen Umschaltung und Aktivierung ihrer Modi, kann nun in Listing 4.5 veranschaulicht werden:

```
1 public class Task extends RealtimeThread {
2     AsynchronouslyInterruptedException jobAIE;
3     AbsoluteTime absoluteDeadline;
4     AbsoluteTime periodBegin;
5     TaskMode activeMode;
6     TaskSchedulerEDF taskScheduler;
7
8     public void activateMode(TaskMode newMode) {
9         jobAIE.fire();
10        activeMode.deactivate();
11        activeMode = newMode;
12        activeMode.activate();
13        setReleaseParameters(newMode.release());
14    }
15
16    void enterBarrier() {
17        setPriority(LogicalPriority.SCHEDULING);
18        taskScheduler.asyncReschedule.fire(); // fire re-scheduling event
19    }
20
21    void leaveBarrier() {
22        periodBegin =
23            periodBegin.add(activeMode.getPeriod());
24        absoluteDeadline =
25            periodBegin.add(activeMode.getDeadline());
26        taskScheduler.reschedule();
27    }
28
29    boolean wFNP() {
30        enterBarrier();
31        boolean ret = waitForNextPeriod();
32        leaveBarrier();
33        return ret;
34    }
35
36    void run() {
37        while (true) {
38            wFNP();
39            int start = measurement.getCpuTime(); // pure cpu time
40            jobAIE.doInterruptible(activeMode); // execute task mode
41            int end = measurement.getCpuTime();
42            activeMode.resSample(end - start); // sample mode's costs (in ms)
43        } } }
```

Listing 4.5: Periodische Task

Ein aktuell aktiver Modus `activeMode` wird periodisch innerhalb der Methode `Task.run()` (Zeile 37) ausgeführt. Er wird durch einen ATC-Mechanismus aufgerufen – dabei wird an einer `AIE`¹¹ `jobAIE` (Zeile 40) die Methode `doInterruptible()` mit dem entsprechenden `Interruptible` Objekt aufgerufen. Diese Methode löst automatisch die `run()` Methode des `Interruptibles` aus wodurch dann die modus-spezifische Task-Logik ausgeführt wird (siehe Listing 4.4). Um den Aufruf von `doInterruptible()` herum wird die reine CPU-Zeit abgetastet, welche für die Ausführung des aktuellen Modus durch die Task benötigt wurde. Die Messung der CPU-Zeit wurde durch einen nativen Zugriff auf den POSIX *CPU Clock* des entsprechenden Echtzeit-Threads realisiert, welchen der Linux Kernel für alle Prozesse intern verwaltet und über POSIX-Schnittstellen zugänglich macht. Der genaue Mechanismus wird weiter unten gezeigt. Dieser Mechanismus ist jedoch nur dann anwendbar, wenn eine RT-JVM eine eins-zu-eins Abbildung zwischen Java- und Betriebssystem-Threads unterstützt, wie im Falle der JamaicaVM.

Die Ausführung eines `Interruptibles` kann beliebig durch „Feuern“ der ihn umschliessenden `AIE` terminiert werden. Zum Beispiel bricht ein Modus-Wechsel mittels `Task.activateMode()` den laufenden Job eines aktuellen `TaskMode` ab (siehe Listing 4.5, Zeile 9) und aktiviert einen neuen Modus, der dann ab der nächsten Task-Periode ausgeführt wird. Der Abbruch eines laufenden Jobs bei einem Modus-Switch ist natürlich optional. Zu guter Letzt müssen die modus-spezifische Periode und Deadline der Task ihrem Zielmodus entsprechend eingestellt werden. Dies wird durch eine Anpassung der Werte der `PeriodicParameters` des entsprechenden Echtzeit-Threads erreicht, welche dann unmittelbar zum darunter liegenden RTSJ Scheduler propagiert werden. Eine Diskussion der Effekte, welche bei einer Perioden- bzw. Deadline-Änderung auftreten können, wird im Kapitel 4.5 fortgesetzt.

Messungen der Task-Kosten pro Periode

Für die Messungen der Task-Kosten pro Periode wurden über das Java Native Interface (JNI) die systemweiten CPU-Zeit Uhren (*cpu time clocks*) innerhalb des Linux-Kernels instrumentiert. Seit der Kernel-Version 2.6.12 und der `glibc` Version 2.2 verwaltet ein Linux System intern für jeden Prozess eine `CLOCK_THREAD_CPUTIME_ID` Uhr vom Typ `clock_t`. Diese gibt die Zeit an, die der Prozess auf einer Recheneinheit verbracht hat. Die CPU-Zeit wird für jeden Prozess seit seiner Erzeugung mitgeführt und ist unbeeinträchtigt von Blockierungseffekten (z.B. Warten auf I/O) oder den Ausführungszeiten anderer Prozesse.

Bevor die Unterstützung der `CLOCK_THREAD_CPUTIME_ID` Uhren im Linux Kernel integriert wurde, wurden diese innerhalb der `glibc` durch die Verwendung prozessor-spezifischer Timer-

¹¹vgl. `AsynchronouslyInterruptedException` in Kapitel 4.2.3

Register, wie z.B. das TSC-Register bei i386 Architekturen, implementiert. Da in einem Multicore- bzw. Multiprozessor-System (SMP) Recheneinheiten (und somit ihre Timer-Register) mit einer unterschiedlichen Verzögerung zueinander starten, konnte diese Implementierung inakurate Werte liefern, wenn Prozesse zwischen unterschiedlichen Kernen migriert wurden. Darüber hinaus kann die Takt-Frequenz der einzelnen CPU-Kerne dynamisch variieren, was eine Abschätzung der CPU-Zeit der Prozesse ohne eine gemeinsame Zeit-Basis weiter erschwert. Die Implementierung der Zeitmessung innerhalb von ARTOS ist jedoch von solchen Problemen unbetroffen, da wie in Kapitel 4.3.3 bereits ausgeführt, alle Prozesse der Echtzeit-Tasks durch entsprechende CPU-Affinitäten auf ein und den selben Rechenkern gesetzt werden. Mit der Einführung der CPU-Zeit Uhren innerhalb des Linux Kernels ab Version 2.6.12 wurden jedoch auch die Probleme bei Multikern-Systemen beseitigt.

Die CPU-Zeit Uhr eines Prozesses kann von jedem anderen Prozess, z.B. über die folgenden POSIX-Schnittstellen ausgelesen werden:

```
1 int pthread_getcpuclockid(pthread_t thread, clockid_t *clock_id);
2 int clock_gettime(clockid_t clk_id, struct timespec *tp);
3
4 struct timespec {
5     time_t    tv_sec;          /* seconds */
6     long      tv_nsec;         /* nanoseconds */
7 };
```

Mit `pthread_getcpuclockid()` wird zunächst die `clock_id` der CPU-Uhr eines Threads ermittelt. Der Thread wird hierbei durch seine POSIX-Thread-ID vom Typ `pthread_t` identifiziert. Mit `clock_gettime()` kann dann die Zeit innerhalb der entsprechenden Uhr abgefragt werden. Diese Zeit wird in einer `timespec`-Struktur mit einer Nanosekunden-Auflösung zurückgeliefert. Die Präzision der Messung hängt jedoch von der jeweiligen Hardware-Architektur und Kernel-Version ab.

```
1 public class Measurement {
2     int cpuClockId = 0;
3     native int getCpuClockId();
4     native long getCpuTime(int clockid);
5
6     public void register() {
7         cpuClockId = getCpuClockId();
8     }
9
10    long getCpuTimeMillis() {
11        return getCpuTime(cpuClockId) / 1000000;
12    }
13 }
```

Listing 4.6: Measurement-Objekt

Bei der Implementierung von ARTOS wurde jede Task mit einem `Measurement` Objekt (siehe Listing 4.6) verknüpft, welches das Auslesen ihrer CPU-Zeit von einem beliebigen Echtzeit-Thread aus ermöglicht. Die Identifikationsnummern der Java `RealtimeThreads` und ihrer `pthread`-Gegenstücke sind voneinander unabhängig und völlig unterschiedlich. Aus diesem Grund wird in ARTOS beim Start einer jeden Task automatisch durch `Measurement.register()` die native ID ihrer CPU-Uhr ermittelt und zwischengespeichert. Innerhalb der nativen Implementierung von `getCpuClockId()`:

```
1 JNIEXPORT jint JNICALL
2 Measurement_getCpuClockId(JNIEnv *env, jobject t) {
3     clockid_t clockid;
4     pthread_getcpuclockid((pthread_t)pthread_self(), &clockid)
5     return (jint)clockid;
6 }
```

wird die `clockid` des aufrufenden Threads anhand seiner durch `pthread_self()` zurückgegebenen `pthread`-ID ermittelt und zurückgegeben. Zu einem beliebigen späteren Zeitpunkt kann für den registrierten `RealtimeThread` die CPU-Zeit durch `Measurement.getCpuTimeMillis()` abgefragt werden. Das native Gegenstück zu `getCpuTime()`:

```
1 JNIEXPORT jlong JNICALL
2 Measurement_getCpuTime(JNIEnv *env, jobject t, jint cpuClockId) {
3     struct timespec tp;
4     clock_gettime((clockid_t)cpuClockId, &tp)
5     long thread_cpu_time = (tp.tv_sec * SEC_IN_NANOSECS) + tp.tv_nsec;
6     return thread_cpu_time;
7 }
```

liest den Wert der angegebenen CPU-Uhr aus und liefert diesen mit einer Nanosekunden-Auflösung innerhalb der JVM zurück. Dieser Wert kann wie in Listing 4.6 innerhalb einer beliebigen anderen Auflösung konvertiert werden. Die aktuelle Implementierung von ARTOS misst die Task-Kosten pro Periode in Millisekunden, was für die meisten anvisierten Anwendungsfälle ausreichend ist.

4.4.3 Applikations-lokales Task-Scheduling

Die Task-Implementierung ist fest verankert mit der untersten Ebene des ARTOS-Scheduling-Modells, d.h. mit der applikations-lokalen Aktivierung der Tasks, welche als nächstes vorgestellt wird. Das Modell schreibt ein Earliest Deadline First-Scheduling auf der Task-Ebene vor, also muss EDF auf der Basis von RTSJ's Fixed-Priority Scheduler emuliert werden. Der hier entwickelte Mechanismus basiert auf der Annahme, dass jede beliebige Scheduling-Policy durch

eine gezielte Manipulation (bzw. Skalierung) von Task-Prioritäten zu bestimmten Zeitpunkten unterstützt werden kann [130]. Auf diese Weise kann dann bestimmt werden, welche Task als nächstes ausgeführt wird. Für EDF wurden zwei Scheduling-Punkte, um den Zeitpunkt der freiwilligen Task-Suspension herum am Ende eines jeden periodischen Jobs, realisiert. Im Rahmen von RTSJ entspricht dieser Punkt der Methode `RealtimeThread.waitForNextPeriod()`. Sie wird typischerweise von einem periodischen Echtzeit-Thread am Ende einer Schleife aufgerufen (siehe `run()` in Listing 4.5), um das Ende seiner Periode zu bestimmen und die Kontrolle freiwillig abzugeben¹². Dabei blockiert der Thread innerhalb des Funktionsaufrufs an einer Sperre, die erst zum Beginn seiner nächsten Periode wieder freigegeben wird.

Ein interessanter Punkt innerhalb der Task-Implementierung in Listing 4.5 ist die Methode `wFNP()`, die als Wrapper für die eigentliche `waitForNextPeriod()` Funktion dient. Damit können Aufrufe an `waitForNextPeriod()` durch entsprechende *enter* und *leave* *Interceptor*-Funktionen¹³ abgefangen werden. Diese Interceptoren werden verwendet, um ein kooperatives EDF-Scheduling durch Manipulation der Task-Prioritäten kurz vor und nach der Task-Suspension zu etablieren. Ein ähnlicher Mechanismus für benutzerdefiniertes Scheduling auf der Anwendungsebene wurde von Zerzelidis et al. in [130, 131] vorgeschlagen. Im Gegensatz dazu ist die in dieser Arbeit vorgestellte Variante durch eine erweiterte API-Ebene über RTSJ realisiert, ohne Anpassungen der Standard RTSJ-Bibliotheken oder der RT-JVM abzuverlangen (für weitere Diskussionen siehe Kapitel 4.5.3). Ein ähnliches Verfahren ist auch auf der Applikations-Ebene des hierarchischen Modells integriert, um dort Scheduling anhand Prioritätsverschiebungen ganzer Anwendungen zu betreiben (das später in Kapitel 4.4.4 vorgestellt wird).

In der hier vorgestellten Implementierung besitzt jede Applikation eine lokale Task-Scheduler Instanz (siehe `TaskSchedulerEDF` in Listing 4.8), welche entweder direkt oder asynchron durch Anwendungs-Tasks aufgerufen wird (vgl. `enterBarrier()` und `leaveBarrier()` in Listing 4.5). Nach einer Deblockierung von `waitForNextPeriod()` rufen Tasks direkt `reschedule()` auf, während vor einer Task-Suspendierung der Scheduler durch ein asynchrones Ereignis `asyncReschedule` ausgelöst wird. Das Scheduling wird dann von einem asynchronen Event-Handler `rescheduleHandler` vollzogen, welcher die selbe Priorität wie die auslösende Task besitzt. Dadurch werden die blockierende Task und die Behandlung des Rescheduling-Events innerhalb der selben FIFO Prioritäts-Liste des `PrioritySchedulers` sequenzialisiert. Dies ist notwendig, da sich anderenfalls die Task durch ein synchrones Rescheduling selbst verdrängen könnte, bevor sie `waitForNextPeriod()` erreicht, was zu einer Fristverletzung führen kann.

¹²innerhalb von `waitForNextPeriod()` ruft die JVM intern `sched_yield()` auf (vgl. Kapitel 4.3.2)

¹³in einem aspekt-orientierten Modell wäre `waitForNextPeriod` ein point-cut

Ein Task-Scheduler benötigt (mindestens) drei *logische Prioritätsstufen*, um das applikations-lokale EDF Scheduling zu vollziehen. Diese Prioritäten werden durch eine Java-Enumeration wie im Code-Listing 4.7 umgesetzt.

```
1 public enum LogicalPriority {  
2     PREEMPTED, ACTIVE, SCHEDULING;  
3 }
```

Listing 4.7: Logische Prioritäten auf der Task-Ebene

Anhand der Ordinal-Zahl ihrer Enumeration-Instanz können die logischen Prioritäten dynamisch auf absolute Prioritäten innerhalb des Prioritätsraums von RTSJ, mit steigenden Werten von **PREEMPTED** bis **SCHEDULING**, abgebildet werden. Die Grundidee ist, dass Anwendungs-Tasks ihre Prioritäten kooperativ zwischen den Prioritätsstufen **PREEMPTED** und **ACTIVE** wechseln, wenn sie gerade den Scheduler aufrufen. Dies geschieht anhand einer künstlichen und dynamisch verwalteten, absoluten Deadline **absoluteDeadline** einer jeden Task (siehe Listing 4.5). Der Wert von **absoluteDeadline** wird innerhalb von **leaveBarrier()** am Anfang einer jeden Task-Periode automatisch aktualisiert. Bevor sich jedoch eine Task in **waitForNextPeriod()** suspendiert, setzt sie ihre eigene Priorität auf **SCHEDULING** hoch. Auf diese Weise wird sichergestellt, dass, wenn sie später innerhalb einer neuen Periode aktiviert wird, sie in der Lage ist instantan die Kontrolle zu übernehmen und den Scheduler innerhalb **leaveBarrier()** erneut aufzurufen (bzw. Scheduling durchzuführen).

Die Modifikation einer RTSJ-Priorität hat eine sofortige Auswirkung. Der betroffene Echtzeit-Thread wird über die **PriorityScheduler**-Abstraktion von RTSJ unmittelbar am Ende der entsprechenden Prioritäts-Warteschlange des unterliegenden Linux FIFO-Schedulers eingefügt (siehe Kapitel 4.3.2). Die applikations-lokale Prioritätsaufteilung für das Task-Scheduling ist in Abbildung 4.3 schematisch dargestellt. Innerhalb einer Anwendung können mehrere Tasks $\tau_{s1}, \tau_{s2}, \dots, \tau_{sn}$ gleichzeitig eine logische **SCHEDULING**-Priorität besitzen. Diese Tasks sind entweder am Ende einer Periode in **waitForNextPeriod()** suspendiert, oder sie führen gerade selbst Scheduling-Aktivitäten aus. Obwohl sie mit einer FIFO-Reihenfolge abgearbeitet werden, werden die Scheduling-Vorgänge an den Methoden einer **TaskSchedulerEDF**-Instanz synchronisiert. Auf diese Weise wird die applikations-lokale Scheduling-Ordnung auch dann beibehalten, wenn Tasks temporär blockieren. Nach dem vorgestellten Scheduling-Prinzip kann zu einem beliebigen Zeitpunkt genau eine Task τ_{a1} die logische Priorität **ACTIVE** besitzen, während mehrere Tasks gleichzeitig mit einer logischen Priorität **PREEMPTED** $\tau_{p1}, \tau_{p2}, \dots, \tau_{pn}$ operieren können. Für eine weitere Diskussion von Blockierungs- und Aktivierungseffekten, die bei dem hier beschriebenen Task-Scheduling Modell auftreten können, siehe Kapitel 4.5.3.

Abbildung 4.3: Prioritätsraum für applikations-lokales Scheduling

Die Implementierung des applikations-lokalen Task-Schedulers ist im folgenden Listing 4.8 dargestellt. Eine `rescheduler` Task, die gerade den lokalen Scheduler aufruft, setzt die Priorität der „wahlfähigsten“ Task auf `ACTIVE` und verdrängt die gerade aktive Task mit einer `PREEMPTED` Priorität. Die Prioritätsumsetzung hat eine sofortige Auswirkung auf die Task-Aktivierungen innerhalb des `PrioritySchedulers` von RTSJ. Entsprechend Earliest Deadline First ist die wahlfähigste Task für eine Aktivierung diejenige, die die kleinste absolute Frist besitzt und ihren aktuellen Job noch nicht fertiggestellt hat. Dieser Prioritäts-Skalierungs-Mechanismus greift auch dann korrekt, wenn die `rescheduler` Task selbst diejenige ist, die aktiviert oder verdrängt werden soll. In `wfnp()` (vgl. Listing 4.5) ruft die Funktion `enterBarrier()` den Scheduler auf und setzt die Priorität des aktuellen Threads auf `SCHEDULING`, bevor sich dieser in `waitForNextPeriod()` selbst suspendiert. In `leaveBarrier()` wird wiederum die Priorität, mit welcher der aktuelle Thread seinen aktuellen Release fortführen wird, dynamisch durch einen synchronen Aufruf des Schedulers entschieden.

Das Ergebnis dieses Scheduling-Mechanismus ist in Abbildung 4.4 exemplarisch dargestellt. Zwei Tasks $\tau_{i,k} = (T, C, D)$, mit einer entsprechenden Periode T , Kosten C und Deadline D werden folgendermaßen definiert: $\tau_{i,1} = (5, 2, 4)$ und $\tau_{i,2} = (3, 1, 2)$. Beide werden durch die selbe Task-Scheduler Instanz gesteuert. Abbildung 4.4 veranschaulicht die entsprechenden Aktivierungen und die Prioritäts-Zustände beider Tasks innerhalb ihrer Hyperperiode. Wie zu sehen ist, suspendieren die Tasks am Ende eines jeden Jobs mit einer `SCHEDULING` Priorität und sind somit in der Lage, bei jeder neuen Aktivierung erneut Rescheduling zu betreiben.

Das Task-Scheduling anhand der festen Prioritäten des RTSJ `PrioritySchedulers` hat die Konsequenz, dass Tasks, die logisch verdrängt sind, trotzdem ausführen können, wenn Tasks mit höheren Prioritäten nicht lauffähig sind. Wenn zum Beispiel, eine Task mit einer `ACTIVE` Priorität temporär blockiert, werden die Tasks mit `PREEMPTED` Prioritäten weiter ausgeführt, bis

die höher-priore Task wieder deblockiert. Dabei stellt der Task-Scheduler logisch trotzdem eine EDF-konforme Task-Aktivierung her. Somit können nieder-priore Tasks von freien Kapazitäten innerhalb einer Anwendung profitieren. Ihre Ausführung ist dabei jedoch nur optimistisch („best-effort“). Ihre Echtzeitgarantien werden allein durch die Aktivierung durch den Task-Scheduler und die Kapazitäts-Bemessung der Anwendung bzw. dem Scheduling auf der Applikations-Ebene berücksichtigt und eingehalten.

```
1 public class TaskSchedulerEDF {
2     AsyncEvent asyncReschedule;
3     Task active;
4
5     AsyncEventHandler rescheduleHandler = new AsyncEventHandler() {
6         public void handleAsyncEvent() {
7             reschedule();
8         }
9     }
10
11     /* return the task with the earliest deadline,
12        that is not suspended in waitForNextPeriod */
13     public synchronized Task getMostEligible() {...};
14
15     public synchronized void reschedule() {
16         Task rescheduler = (Task)Thread.currentThread();
17         Task next = getMostEligible();
18
19         if(active == next) {
20             // rescheduler does not preempt
21             if(rescheduler != next)
22                 // block rescheduler if it is not next
23                 rescheduler.setPriority(LogicalPriority.PREEMPTED);
24         } else {
25             // rescheduler preempts the currently active
26             active.setPriority(LogicalPriority.PREEMPTED);
27
28             // rescheduler sets (evtl. itself as) new active
29             active = next;
30
31             if (rescheduler == next) {
32                 // set ACTIVE prio
33                 rescheduler.setPriority(LogicalPriority.ACTIVE);
34             } else {
35                 // set active prio for next
36                 next.setPriority(LogicalPriority.ACTIVE);
37                 // rescheduler has to block itself
38                 active.setPriority(LogicalPriority.PREEMPTED);
39             } } } }
```

Listing 4.8: Task-Scheduler

Abbildung 4.4: EDF-Aktivierung mit Prioritätsverschiebungen (aus [100])

4.4.4 Implementierung und Scheduling von CBS

Wie in Kapitel 3.2.2 bereits geschildert, werden Applikationen und ihre Tasks innerhalb Constant Bandwidth Servern gekapselt. Auf diese Weise werden die Ressourcenverbräuche der Anwendungen voneinander isoliert, während sie in ihren aktuellen Modi ausgeführt werden. Die von jeder Anwendung, für eine erfolgreiche Ausführung auf der Task-Ebene, benötigte Server-Kapazität $R_{i,j}$, wird dynamisch mit den aktuellen Schätzungen der Kosten und der Rechenzeitanforderungen ihrer Tasks berechnet. Wie in Kapitel 3.2.2 gezeigt wurde, wird diese Kapazität im Zuge einer applikations-lokalen Planbarkeitsanalyse für eine modus-abhängige Server-Periode $P_i = \min(T_{i,k})$ bestimmt. Die Periode kann für jeden Anwendungsmodus unterschiedlich ausfallen, da sich die Task-Definitionen zwischen den Modi unterscheiden können. Somit muss bei jedem Modus-Wechsel einer Anwendung die Kapazität und die Periode ihres CBS ihrem Zielmodus entsprechend eingestellt werden.

Darüber hinaus finden Kapazitätsaktualisierungen statt, wenn Applikations-Tasks ihre jitter-abhängigen Sicherheitspuffern verletzen (siehe Abbildung 3.10). In solchen Fällen weicht die aktuelle kumulative Rechenzeitanforderung der gesamten Anwendung von der erwarteten ab und

eine neue Server-Kapazität muss ermittelt werden. Dadurch verändert sich die Bandbreiten-Allokation der Anwendung, was zufolge hat, dass die aktuelle Gesamtkonfiguration der Anwendungsmodi im System möglicherweise nicht mehr optimal ist. In diesem Fall wird auf einer höheren Ebene des Modells eine Optimierung (siehe Kapitel 3.2.3) angestoßen, die anhand der ermittelten Bandbreiten der Modi über eine neue optimale Konfiguration entscheidet. Dabei werden potentielle Engpässe sowie freie Kapazitäten für höhere Anwendungsmodi berücksichtigt.

Für die Realisierung einer Lösung für Constant Bandwidth Server mit RTSJ, sind spezielle Funktionen auf der RTSJ-Ebene notwendig, die durch die aktuelle Version der Spezifikation 1.0.2 und einer Standard RT-JVM zunächst nicht unterstützt werden. Auf der Basis der JamaicaVM konnten jedoch Anpassungen bestimmter RTSJ-Mechanismen und ihrer API gemacht werden, mit denen eine Implementierung des Modells möglich war. In der im Folgenden präsentierten Lösung werden somit auch Erweiterungen der RTSJ-Spezifikation vorgeschlagen, die allgemein für die Implementierung benutzerdefinierter Server-Mechanismen mit RTSJ hilfreich sind. Einige dieser Erweiterungen wurden bereits in [100] publiziert und im Rahmen der RTSJ Expert Group diskutiert. Manche von ihnen wurden innerhalb der neuen Spezifikation RTSJ 2.0 berücksichtigt, was später genauer ausgeführt wird.

4.4.4.1 Realisierung von CBS

In dieser Arbeit wurde eine Lösung für Constant Bandwidth Servern entwickelt, die zunächst auf die RTSJ Cost-Monitoring-Funktionalität für PGPs aufsetzt (siehe Kapitel 4.2.4). In Listing 4.9 ist eine primäre Definition eines CBS dargestellt. Diese wird im weiteren Verlauf des Kapitels sukzessive mit neuen Funktionen erweitert.

```
1 public class CBS extends ProcessingGroupParameters {
2     Application app;
3     RelativeTime capacity;
4     AbsoluteTime absoluteDeadline;
5
6     public void reconfigure() {
7         capacity = app.activeMode.getRequiredCapacity();
8         setCost(capacity);
9         setPeriod(app.activeMode.getMinTaskPeriod());
10    }
11    ...
12 }
```

Listing 4.9: Constant Bandwidth Server-Implementierung (1)

Ein CBS ist eine Spezialisierung einer Prozessgruppe und kann somit innerhalb der Kostenüberwachung einer RT-JVM integriert werden. Er besitzt eine Referenz auf die gekapselte Applikation `app`, eine applikationsmodus-abhängige, maximale Kapazität `cbsCapacity` und eine absolute Scheduling-Deadline `absoluteDeadline`. Bei der Initialisierung einer Anwendung werden alle Applikations-Tasks mit der entsprechenden CBS-Instanz ihrer Anwendung verknüpft und mit einem gemeinsamen Start-Zeitpunkt versehen (synchroner Start der Tasks).

Wie in Kapitel 4.4.1 bereits gezeigt wurde, wenn die Anwendung in einem bestimmten Modus aktiviert oder später gewechselt wird, wird in einer privaten Variable `Application.activeMode` eine Referenz auf ihren aktuellen Modus zwischengespeichert. Nach jeder Modus-Veränderung wird anschließend automatisch der CBS der Anwendung mit `reconfigure()` (vgl. Listing 4.3) neu eingestellt. Die Implementierung von `CBS.reconfigure()` ist nun im Listing 4.9 zu sehen. Hierbei werden zunächst eine modus-abhängige Server-Kapazität und -Periode eingestellt. Es ist zu beachten, dass `setCost()` und `setPeriod()` die Eigenschaften des dem CBS zugrundeliegenden PGP-Objekts verändern. Dabei wird mit `setCost` die maximale Kapazität, mit welcher der Server aufgeladen werden soll, angegeben. Nach der RTSJ-Spezifikation gelten die neuen Werte erst ab der darauffolgenden PGP-Periode.

Mit `getMinTaskPeriod()` wird die Server-Periode gleich der kleinsten Task-Periode innerhalb des jeweiligen Anwendungsmodus gesetzt (vgl. Kapitel 3.2.2). Mit `getRequiredCapacity()` wird die zuletzt ermittelte modus-spezifische Rechenzeitanforderung der Tasks zugewiesen, welche bei Veränderungen der Task-Approximationen, durch eine applikations-lokale Planbarkeitsanalyse, ermittelt und normiert wird (vgl. Kapitel 3.2.2).

Obwohl RTSJ mit den PGPs einen Basis-Mechanismus zur Überwachung der Kosten einer Gruppe von Echtzeit-Threads liefert, ist dieser zunächst ungeeignet für die Implementierung von CBS. Im Gegensatz zu einer Processing Group, wird die Kapazität eines Constant Bandwidth Servers sofort aufgefüllt, wenn sie vollständig aufgebraucht ist (vgl. Kapitel 3.2.2), und nicht periodisch. Dabei wird auch die Scheduling-Deadline des Servers um eine Periode verlängert. Für PGPs existiert in RTSJ kein Scheduling-Mechanismus, anhand welchen das hierarchische Scheduling von ARTOS umgesetzt werden kann. Die PGPs werden in der Regel durch Timer implementiert, mit denen die Kosten der Threads in bestimmten Zeitabständen überprüft werden. Der Vorgang speziell innerhalb der JamaicaVM ist im Kapitel 4.5.2 grob beschrieben. Die Aktivierung der mit einem PGP-Objekt verbundenen Threads, wird wiederum allein durch ihre Perioden und Prioritäten auf der Ebene des `RTSJ.PrioritySchedulers` gesteuert. Für das ARTOS-Modell wird jedoch EDF-Scheduling auf der CBS-Ebene benötigt. Dieses soll dynamisch den nächsten auszuführenden Server anhand seiner Scheduling-Deadline auswählen und aktivieren können, während andere Server temporär suspendiert bzw. verdrängt werden.

4.4.4.2 CBS-Scheduling durch Prioritätsverschiebung ganzer Anwendungen

Das EDF-Scheduling der CBS wird in einer ähnlichen Weise wie auf der Task-Ebene durch Skalierung von Aktivitäts-Prioritäten innerhalb des Prioritätsraums von RTSJ emuliert. Dieses Mal werden jedoch ganze Anwendungen mit ihren internen Tasks dynamisch durch einen system-weiten **CBSScheduler** priorisiert. Für die Scheduling-Aktivitäten gelten hier auch andere Voraussetzungen als auf der Task-Ebene. Es werden außerdem nur zwei Prioritäts-Ebenen benötigt, da die CBS selbst keine Aktivitäten sind, die ein kooperatives Scheduling innerhalb einer expliziten **SCHEDULING**-Priorität betreiben können.

Die Scheduling-Vorgänge auf der Applikations-Ebene werden ausschließlich durch asynchrone Ereignisse aus unterschiedlichen Ebenen des Modells ausgelöst. Die Vorgänge selbst werden innerhalb einer **CBSScheduler**-Instanz durch entsprechende asynchrone Event-Handler (AEH) implementiert und ausgeführt. Manche dieser Ereignisse werden automatisch von einer RTSJ-konformen RT-JVM unterstützt, andere mussten explizit bereitgestellt werden. Die genauen Stellen und Bedingungen für das Auslösen der Scheduling-Events werden im Folgenden erklärt. Als nächstes wird das Prinzip der Aktivierung bzw. Deaktivierung der Server (und somit ganzer Anwendungen) durch Verschiebungen im RTSJ-Prioritätsraum erläutert.

Abbildung 4.5: Aufteilung des Prioritätsraums von ARTOS

Die für das CBS-Scheduling nötige Aufteilung des RTSJ-Prioritätsraums wird in Abbildung 4.5 schematisch dargestellt. Die CBS werden vom **CBSScheduler** anhand ihrer Server-Deadlines **absoluteDeadline** zwischen den *Prioritäts-Bändern* **ACTIVE** und **PREEMPTED** verschoben. Jedes Band ist realisiert durch drei logische Task-Prioritäten, die für das applikations-lokale Scheduling

auf der Task-Ebene notwendig sind (vgl. Listing 4.7). Wenn zum Beispiel der CBS einer Anwendung zu einem beliebigen Zeitpunkt aktiviert wird, dann werden die absoluten Prioritäten seiner internen Tasks anhand ihrer aktuellen logischen Prioritäten relativ zum **ACTIVE** Prioritäts-Band eingestellt (siehe Abbildung 4.5). Wird ein CBS wiederum suspendiert, dann bekommen seine internen Tasks absolute Prioritäten innerhalb des **PREEMPTED** Prioritäts-Bands zugewiesen. Zu jedem Zeitpunkt ist nur eine Anwendung aktiv, die Server aller anderen Anwendungen sind mit ihren Tasks innerhalb des niedrigeren Prioritäts-Bands verdrängt.

Über den Prioritäts-Bändern der Anwendungen wird eine zusätzliche **EVENTHANDLING**-Priorität für die Event-Handler (AEH) des **CBSschedulers** reserviert. Mit dieser Priorität werden alle Scheduling-Events auf der Applikations-Ebene abgefangen und abgearbeitet. Auf diese Weise wird sichergestellt, dass der **CBSscheduler** die applikations-lokalen Task-Aktivierungen unterbrechen und verdrängen kann – das CBS-Scheduling hat also eine höhere Priorität als das Task-Scheduling. Außerdem werden die Ausführungs-Instanzen der AEHs innerhalb derselben FIFO-Prioritäts-Warteschlange des **PrioritySchedulers** eingereiht und somit parallele Event-Verarbeitungen synchronisiert. Die **GLOBAL_SCHEDULING**-Priorität wird für die Server-Aktivierungen benötigt und wird später näher erläutert.

```
1 public enum PriorityBand {
2     PREEMPTED(20), ACTIVE(22);
3
4     public final int basePriority;
5
6     private PriorityBand (int basePriority) {
7         this.basePriority = basePriority;
8     }
9
10    public void assign (CBS cbs) {
11        // assign the new band to the CBS
12        cbs.priorityBand = this;
13
14        // re-scale priorities of wrapped tasks
15        for (Task task : cbs.getApp().getTasks()) {
16            task.changePriorityBand(this);
17        } } }
```

Listing 4.10: Realisierung der Prioritäts-Bänder

Die Implementierung der Prioritäts-Bänder wurde anhand einer Java-Enumeration wie in Listing 4.10 realisiert. Die Enumeration-Instanzen **ACTIVE** und **PREEMPTED** repräsentieren die zwei Bänder der Anwendungen. Sie werden mit entsprechenden Basis-Prioritäten **basePriority** für die Abbildung von logischen Task-Prioritäten auf absolute RTSJ-Prioritäten initialisiert.

Die Prioritäts-Abbildung findet dann dynamisch statt, wenn durch `assign()` ein CBS einem Prioritäts-Band – also einer Enumeration-Instanz – zugewiesen wird. Dabei werden mit `task.changePriorityBand(this)` allen gekapselten Anwendungs-Tasks, anhand ihrer logischen Prioritäten, neue absolute Prioritäten innerhalb des entsprechenden Prioritäts-Bands vergeben.

Für die Realisierung der Prioritäts-Bänder ist eine Erweiterung der Task-Definition aus Code-Listing 4.5 notwendig, welche im folgenden Listing 4.11 aufgeführt ist. Jede Task besitzt nun zwei Variablen `logicalPriority` und `band`, die ihre aktuelle logische Priorität und das aktuelle Prioritäts-Band ihres Servers angeben. Durch die Helfer-Funktion `setPriority()` kann nun der Task eine neue logische Priorität innerhalb ihres aktuellen Bands zugewiesen werden. Ihre absolute Priorität wird dabei durch den Ordinal-Wert der übergebenen `LogicalPriority`-Instanz berechnet und gesetzt. In der Funktion `changePriorityBand()` wird auf ähnlicher Weise das Prioritäts-Band einer Task gewechselt, wobei hier ihre relative, logische Priorität unverändert bleibt.

```
1 public class Task extends RealtimeThread {
2     ...
3     public LogicalPriority logicalPriority;
4     public PriorityBand band;
5
6     public void setPriority(LogicalPriority logicalPriority) {
7         PriorityParameters pp = (PriorityParameters)this.getSchedulingParameters();
8         pp.setPriority(band.basePriority + logicalPriority.ordinal());
9         this.logicalPriority = logicalPriority;
10    }
11
12    public void changePriorityBand(PriorityBand band) {
13        PriorityParameters pp = (PriorityParameters)this.getSchedulingParameters();
14        pp.setPriority(band.basePriority + logicalPriority.ordinal());
15        this.band = band;
16    }
17 }
```

Listing 4.11: Erweiterung der Task-Definition (1)

Anhand der Definition der Prioritäts-Bändern und den entsprechenden Erweiterungen auf der Task-Ebene, kann nun der `CBSScheduler`, wie in Listing 4.12, implementiert werden. Der `CBSScheduler` verwaltet eine interne Liste mit Referenzen auf die im System residierenden CBS. Ein Scheduling-Vorgang wird über ein asynchrones Ereignis `rescheduleEvent` ausgelöst und innerhalb eines entsprechenden Event-Handlers `rescheduleEventHandler` abgefangen und behandelt. Im Konstruktor des `CBSSchedulers` wird der Handler mit einer entsprechen-

den EVENTHANDLING-Priorität initialisiert und registriert. Ein Scheduling-Event kann dann mit `rescheduleEvent.fire()` von einer beliebigen Stelle innerhalb des Frameworks ausgelöst werden (vgl. Kapitel 4.2.3, „Asynchrone Ereignisverarbeitung“).

```
1 public final class CBSScheduler {
2     CBS active = null;
3     ArrayList<CBS> servers = new ArrayList<CBS>();
4     public AsyncEvent rescheduleEvent = new AsyncEvent();
5
6     AsyncEventHandler rescheduleEventHandler = new AsyncEventHandler() {
7         public void handleAsyncEvent() { reschedule(); };
8     };
9
10    public CBSScheduler() {
11        rescheduleEventHandler.setSchedulingParameters(EVENTHANDLING);
12        rescheduleEvent.addHandler(rescheduleEventHandler);
13    }
14
15    CBS getMostEligible() {
16        CBS next = null;
17        Collections.sort(servers, null);
18        Iterator<CBS> it = servers.iterator();
19        while (it.hasNext()) {
20            next = it.next();
21            if(next.absoluteDeadline.compareTo(now) > 0) break;
22        }
23        return next;
24    }
25
26    synchronized void reschedule() {
27        CBS next = getMostEligible();
28        if(active == next) return;
29        // preempt currently active
30        if(active != null) PriorityBand.PREEMPTED.assign(active);
31        // activate next
32        if(next != null) PriorityBand.ACTIVE.assign(next);
33        active = next;
34    }
35 }
```

Listing 4.12: Implementierung des CBSSchedulers

Bei der Ereignis-Behandlung innerhalb des Event-Handlers wird letztendlich in `reschedule()` die EDF-Aktivierung der Anwendungs-Server durchgeführt. Dabei wird zunächst in `getMostEligible()` der CBS mit der kleinsten Scheduling-Deadline `absoluteDeadline` ermittelt. Hierfür wird die Liste der CBS anhand ihrer internen Deadlines in aufsteigender Reihenfolge sortiert. Das erste Element ist dann der zu aktivierende Server mit der kleinsten, absoluten

Deadline. Der Server wird dann innerhalb des **ACTIVE** Prioritäts-Bands verschoben, während ein bislang aktiver CBS innerhalb des **PREEMPTED** Bands verdrängt wird.

Die absoluten Prioritäten der Tasks im RTSJ-Prioritätsraum werden nun durch beide Scheduler auf der Task- und auf der Applikations-Ebene bestimmt. Entscheidend dafür ist der Zustand und die Deadline ihrer CBS, sowie ihre eigenen Aktivierungszustände und absolute Fristen. Ähnlich wie schon im Kapitel 4.4.3 für nieder-priore Aktivitäten auf der Task-Ebene gezeigt wurde, können hier ganze Anwendungen von freien Kapazitäten im System instantan profitieren. Solange die Tasks einer aktiven Anwendung temporär blockieren oder sich suspendieren, können Tasks von verdrängten Anwendungen ausgeführt werden. Ihre Aktivierung ist jedoch nicht garantiert oder vorhersehbar. Sie ergibt sich zur Laufzeit durch die Reihenfolge, mit der die logischen Prioritäten des **PREEMPTED** Prioritäts-Bands durch die Anwendungen belegt werden. In den entsprechenden Prioritäts-Warteschlangen residieren jetzt die Tasks aller verdrängten Anwendungen. Sie werden optimistisch in einer strikten, prioritäts-basierten FIFO-Reihenfolge abgearbeitet. Ihre Echtzeitgarantien werden jedoch allein durch die Prioritäts-Skalierungs-Mechanismen auf der Task- und auf der Applikations-Ebene berücksichtigt und eingehalten.

4.4.4.3 Definition von Scheduling-Punkte und Bedingungen

Während die Scheduling-Punkte auf der Task-Ebene um die periodischen Task-Suspendierungen herum mit `waitForNextPeriod()` realisiert werden konnten, unterstützt RTSJ eine solche Semantik auf der PGP-Ebene nicht. Um ein CBS-Scheduling anhand Prioritäts-Skalierung auf einer ähnlichen Weise zu ermöglichen, müssen zunächst die Punkte identifiziert werden, an denen potentiell Veränderungen der Scheduling-Deadline eines Servers auftreten können. An diesen Punkten ist eine Neuplanung und entsprechende Prioritäts-Skalierung der Anwendungs-Server notwendig. Nach den CBS-Regeln auf Seite 89 ist eine Aktualisierung der Server-Deadline jedoch abhängig vom aktuellen Zustand („aktiv“ bzw. „inaktiv“) eines Servers. Aus diesem Grund müssen die Zustandsänderungen eines Servers ebenfalls mitverfolgt werden. Anhand dieser Bedingungen wurden zunächst drei Reaktionspunkte identifiziert, an denen ein Vorgang auf der Applikations-Ebene notwendig ist:

1. *Server-Aktivierung* - bei der Aktivierung einer Task innerhalb eines inaktiven Servers
2. *Server-Deaktivierung* - bei der Suspendierung einer Task ohne ausstehende Jobs anderer Tasks
3. *Kapazitäts-Entleerung* - wenn die Server-Kapazität verbraucht ist

Dementsprechend finden Prioritätsverschiebungen der Anwendungen nur an den Punkten statt, bei denen Veränderungen ihrer Server-Deadlines auftreten, d.h. potentiell bei einer Server-Aktivierung (Punkt 1) und bei einer Kapazitäts-Entleerung (Punkt 3). Die Vorgänge bei einer Server-Deaktivierung (Punkt 2) werden nur zur Aktualisierung des Server-Zustands benötigt.

Die für das CBS-Scheduling notwendigen Bedingungen und Vorgänge an den oben beschriebenen Reaktionspunkten, werden im Folgenden näher vorgestellt. Zur Verwaltung des aktuellen Zustands eines Servers, gemäß der CBS-Definition auf Seite 89, wird seine Implementierung aus Listing 4.9 zunächst mit einer Zustandsvariablen `active` erweitert. Das Ergebnis ist in Listing 4.13 dargestellt. Zu Beginn ist ein CBS stets *inaktiv*.

```

1 public class CBS extends ProcessingGroupParameters {
2     Application app;
3     RelativeTime capacity;
4     Optimization optimization;
5     AbsoluteTime absoluteDeadline;
6
7     boolean active = false;
8
9     public void reconfigure() {
10         cbsCapacity = app.activeMode.getRequiredCapacity();
11         setCost(cbsCapacity);
12         setPeriod(app.activeMode.getMinTaskPeriod());
13     }
14     ...
15 }
```

Listing 4.13: Constant Bandwidth Server-Implementierung (2)

1. Server-Aktivierung

Wird eine Task zu einem Zeitpunkt $r_{i,j,k}$ innerhalb eines inaktiven Servers (periodisch) aktiviert, dann muss nach CBS-Regel 7 überprüft werden, ob ihr eintreffender Job mit der aktuellen Kapazität und Deadline des Servers bedient werden kann. Falls ja, bleiben die CBS-Parameter unverändert. Eine Neuplanung auf der Serverebene findet nur dann statt, wenn der aktivierte CBS die kürzeste Deadline besitzt. Falls jedoch Regel 7 greift, dann muss in jedem Fall ein Rescheduling durchgeführt werden. Eine Überprüfung dieser Bedingungen muss darum zum Beginn einer jeden Task-Periode vollzogen werden. Der genaue Vorgang wird durch die folgenden Punkte vorgegeben:

- Überprüfe ob Server inaktiv (also „idle“) ist, falls nein findet kein Rescheduling statt. Falls ja, setze den Zustand des Servers auf aktiv und fahre mit den folgenden Schritten fort.
- Falls $c_s < (D_{i,j} - r_{i,j}) \cdot B_{i,j}$ überprüfe ob der Server die kleinste Scheduling-Deadline hat. Falls ja wird der Server im ACTIVE-Prioritäts-Band verschoben und eine aktuell dort aktive Anwendung ggf. verdrängt. Dabei werden die internen Tasks des Servers mit seiner aktuellen Kapazität und Deadline bedient.
- Falls $c_s \geq (D_{i,j} - r_{i,j}) \cdot B_{i,j}$, dann wird die Server-Kapazität $R_{i,j}$ voll aufgeladen, eine neue Server-Deadline $D_{i,j} = r_{i,j} + P_{i,j}$ gesetzt und anschliessend der CBSScheduler aufgerufen.

Im theoretischen Aktivierungsbeispiel zweier Anwendungen mit CBS in Abbildung 3.15 auf Seite 100, findet zum Beispiel eine Server-Aktivierung für A_1 zu den Zeitpunkten $t = 0$ und $t = 15$ statt. Bei $t = 0$ ist der Server mit einer Restkapazität von $r_1 = 0$ noch inaktiv und seine Scheduling-Deadline noch nicht definiert. Da hier CBS-Regel 7 greift, wird eine initiale Deadline $D_{1_1} = 0 + P_1$ gesetzt und die Server-Kapazität mit $c_1 = R_1$ voll aufgeladen. Bei $t = 14$ suspendiert sich die Task τ_2 nach einer Kapazitäts-Entleerung des Servers und hinterlässt diesen in einem inaktiven Zustand mit voller Kapazität und der Server-Deadline D_{1_6} . Wenn dann zum Zeitpunkt $t = 15$ der nächste Job von τ_2 eintrifft, wird der Server erneut aktiviert (die CBS-Regel 7 greift dann jedoch nicht). Die Task wird dann mit der aktuellen Server-Kapazität und -Deadline ausgeführt.

Wie bereits erklärt, muss eine potentielle Server-Aktivierung bei der periodischen Aktivierung einer jeden Task überprüft werden. Aus diesem Grund wurde dafür eine Implementierung auf der Task-Ebene bereitgestellt. Damit die Tasks die Bedingungen ihrer Server überprüfen können, wurden sie so modifiziert, dass sie sich am Ende ihrer Perioden mit einer globalen Scheduling-Priorität `GLOBAL_SCHEDULING` in `waitForNextPeriod` suspendieren. Diese Priorität ist höher als die `EVENTHANDLING`-Priorität für die asynchronen Aufrufe des CBS-Schedulers (vgl. Abb. 4.5). Auf diese Weise können bei jeder Task-Aktivierung kurzfristig alle Vorgänge auf der Task- und Applikations-Ebene unterbrochen und gegebenenfalls der CBS-Scheduler aufgerufen werden, ohne die Tasks längerfristig zu verzögern. In Listing 4.14 wird die entsprechende Erweiterung der Task-Definition aus Kapitel 4.4.2 (im Listing 4.5) vorgestellt. Jede Task bekommt zunächst eine Referenz auf den entsprechenden CBS ihrer Anwendung und auf den system-weiten CBS-Scheduler. Kurz vor ihrer periodischen Suspendierung erhöht die Task in `enterBarrier()` ihre Priorität auf `GLOBAL_SCHEDULING`. Somit wird der Code der `leaveBarrier()` Methode bis Zeile 23 mit der globalen Priorität ausgeführt. Danach wird die Task-Priorität auf die `SCHEDULING`-Priorität im aktuellen Prioritäts-Band ihrer Anwendung abgesenkt, bevor der Task-Scheduler aufgerufen wird.

```
1 public class Task extends RealtimeThread {
2     ...
3     CBS cbs = app.cbs;
4     CBSScheduler cbsScheduler;
5
6     void leaveBarrier() {
7         periodBegin = periodBegin.add(activeMode.getPeriod());
8         absoluteDeadline = periodBegin.add(activeMode.getDeadline());
9
10        if (!cbs.active) {
11            RelativeTime remainingCapacity = cbs.getRemainingBudget();
12
13            if (remainingCapacity >=
14                (cbs.absoluteDeadline - periodBegin) * (cbs.capacity / cbs.getPeriod())){
15                cbs.setRemainingBudget (cbs.capacity);
16                cbs.absoluteDeadline = periodBegin + cbs.getPeriod();
17                cbsScheduler.rescheduleEvent.fire();
18            }
19
20            cbs.active = true;
21        }
22
23        setPriority(LogicalPriority.SCHEDULING, band);
24        taskScheduler.reschedule();
25    }
26
27    void enterBarrier() {
28        setPriority(LogicalPriority.SCHEDULING);
29        taskScheduler.asyncReschedule.fire();
30        setPriority(GLOBAL_SCHEDULING);
31    }
32 }
```

Listing 4.14: Erweiterung der Task-Definition (2)

Betrachten wir zunächst die Vorgänge in `leaveBarrier()`. Nach einer Aktualisierung des Aktivierungszeitpunkts `periodBegin` und der Deadline `absoluteDeadline` der Task, wird der aktuelle Server-Zustand und die Bedingung der CBS-Regel 7 überprüft. Eine Aktualisierung der Server-Deadline (Zeile 16) kann zu einer potentiellen Verdrängung der Anwendung und zur Aktivierung eines anderen CBS führen. Da die Scheduling-Vorgänge auf der Applikations-Ebene nicht durch die Vorgänge auf der Task-Ebene behindert werden dürfen, wird in Listing 4.14 der applikations-lokale `taskScheduler` nach dem `cbsScheduler` aufgerufen. Dabei wird in Zeile 17 das `rescheduleEvent` ausgelöst, in dessen Handler die system-weite `CBSScheduler`-Instanz ausgeführt wird. Wie bereits erklärt, laufen diese Handler mit einer höheren Priorität als die Anwendungen. Infolgedessen kann eine sich gerade in `leaveBarrier` befindende Task nach ihrer

Prioritäts-Absenkung in Zeile 23 samt ihrer Anwendung durch den CBS-Scheduler verdrängt und zu einem späteren Zeitpunkt wieder aktiviert werden.

Im Listing 4.14 werden zwei Funktionen, `getRemainingBudget()` und `setRemainingBudget()`, aufgerufen, mit denen die verbleibende Server-Kapazität abgefragt und bestimmt werden kann. Diese Funktionalität ist grundlegend für die Realisierung von CBS. Die Implementierung dieser Funktionen ist jedoch problematisch, da die Kostenüberwachung der CBS auf der Basis der RTSJ PGP's implementiert wurde. RTSJ stellt jedoch keine Standard-Mechanismen zur Verfügung, mit denen der interne Zustand einer Prozess-Gruppe inspiziert oder modifiziert werden kann. Diese Problematik wird später im Kapitel 4.5.2 weiter erörtert. Dort wird ebenfalls eine proprietäre Übergangslösung vorgestellt, welche mit der JamaicaVM entwickelt wurde. Das Problem der benutzerdefinierten Kapazitäts-Bestimmung für eine Prozess-Gruppe, wurde bereits in [100] veröffentlicht und mit der RTSJ Expert Group in Hinblick auf die anstehende RTSJ Version 2.0 diskutiert.

2. Server-Deaktivierung

Mit der Deaktivierung eines Servers wird der Fall beschrieben, bei dem eine aktuell ausgeführte Task innerhalb einer Anwendung sich selbst suspendiert und keine weiteren aktiven Tasks mit ausstehenden Jobs vorhanden sind. Nach CBS-Regeln 5 und 8 wechselt der Server dann in einen inaktiven Zustand mit einer Restkapazität c_s . Ein Beispiel wurde bereits weiter oben für das Aktivierungs-Szenario in Abbildung 3.15 gezeigt. Zum Zeitpunkt $t = 14$ suspendiert sich die Task τ_2 nach einer Kapazitäts-Entleerung des Servers von A_2 und hinterlässt diesen in einem inaktiven Zustand mit voller Kapazität und der Server-Deadline D_{16} .

Dieser Fall muss ebenfalls auf der Task-Ebene abgefangen und der entsprechende Server dabei als inaktiv markiert werden. Eine Implementierung wurde innerhalb der `Task.enterBarrier()` Methode etabliert (siehe Listing 4.15), welche das Task-Scheduling vor `waitForNextPeriod()` implementiert. Dabei traversiert eine sich selbst suspendierende Task über alle Tasks innerhalb ihrer eigenen Anwendung und überprüft, ob diese bereits in `waitForNextPeriod()` warten. Wenn dies der Fall ist, wird der Server als inaktiv markiert. Dies muss wieder vor der Aktivierung des Task-Schedulers passieren, da sonst die Zustandsänderung des CBS verzögert werden kann.

Mit `isWaitingForNextPeriod()` in Zeile 10 wird wieder eine Funktion zur Abfrage des Suspendierungszustands eines `RealtimeThread` benötigt, die durch RTSJ nicht unterstützt wird. Da die JamaicaVM diesen Zustand für jeden Thread verwaltet, war es einfach diesen über Java-Reflection Mechanismen innerhalb der JVM abzugreifen. Eine andere Lösung wäre eine

Flag-Variable `isSuspended` für jede Task explizit zu verwalten, die im `enterBarrier` gesetzt und in `leaveBarrier` entsprechend gelöscht wird.

```
1 public class Task extends RealtimeThread {
2     ...
3     void enterBarrier() {
4         setPriority(LogicalPriority.SCHEDULING);
5
6         boolean suspending = true;
7
8         for (Task task : app.getTasks()) {
9             if (this == task) continue;
10            if (!task.isWaitingForNextPeriod()) {
11                suspending = false;
12                break;
13            }
14        }
15
16        if (suspending) cbs.active = false;
17        taskScheduler.asyncReschedule.fire();
18        setPriority(GLOBAL_SCHEDULING);
19    }
20    ...
21 }
```

Listing 4.15: Erweiterung der Task-Definition (3)

3. Kapazitäts-Entleerung

Wenn die Kapazität eines Servers mit $c_s = 0$ vollständig aufgebraucht ist, wird diese nach der CBS-Regel 4 wieder vollständig aufgefüllt und die Server-Deadline um eine Periode nach hinten verschoben. Beim Aktivierungs-Szenario in Abbildung 3.15 tritt dieser Fall häufiger auf, für den CBS von A_1 zum Beispiel zu den Zeitpunkten $t = 1.6$, $t = 5.2$, $t = 6.8$ usw. Da die Deadline-Verschiebung einer Prioritätsverminderung gleicht, muss die Aktivierungsreihenfolge der Server neu überprüft werden. Unter Umständen führt dies zu einer Verdrängung des aktuell aktiven Servers. Da die PGPs bereits eine Methode zur Kostenüberwachung bereitstellen, kann diese hier verwendet werden. Dabei wird bei einem Kosten-Überlauf automatisch ein asynchroner Event-Handler `overrunHandler` durch die RT-JVM aufgerufen, welcher dann die folgenden Schritte betreibt:

- Inkrementiere die aktuelle Server-Deadline mit einer Server-Periode $D_{i,j} = D_{i,j} + P_{i,j}$.
- Fülle die Server-Kapazität wieder vollständig auf $c_s = R_{i,j}$.
- Rufe den `CBSScheduler` auf.

Die entsprechende Erweiterung der Implementierung eines CBS ist in Listing 4.16 aufgeführt:

```
1 public class CBS extends ProcessingGroupParameters {
2     ...
3     CBSScheduler cbsScheduler;
4
5     AsyncEventHandler overrunHandler =
6         new AsyncEventHandler() {
7             public void handleAsyncEvent() {
8                 absoluteDeadline = absoluteDeadline + getPeriod();
9                 setRemainingBudget (cbs.capacity);
10                cbsScheduler.reschedule();
11            } }
12
13     public CBS() {
14         setCostOverrunHandler(overrunHandler);
15         overrunHandler.setSchedulingParameters(EVENTHANDLING)
16     }
17     ...
18 }
19 }
```

Listing 4.16: Constant Bandwidth Server-Implementierung (3)

Der Überlauf-Handler `overrunHandler` eines CBS wird mit einer `EVENTHANDLING`-Priorität initialisiert und innerhalb des zugrunde liegenden PGP-Objekts registriert. Somit wird die CBS-Instanz automatisch innerhalb der Kostenüberwachung der JVM integriert. Die Kostenüberwachung richtet sich dann nach den durch `setRemainingBudget()` gesetzten Werten für jeden CBS. Wie bereits erwähnt, wird das Problem einer benutzerdefinierten Kapazitäts-Bestimmung für eine RTSJ Prozess-Gruppe weiter im Kapitel 4.5.2 diskutiert.

Überlauf-Ereignisse werden nun mit derselben Priorität wie die Rescheduling-Ereignisse innerhalb des `CBSScheduler`s verarbeitet (vgl. Listing 4.12). Aus diesem Grund kann der Überlauf-Handler eines CBS selbst den `CBSScheduler` ausführen (vgl. Zeile 10), oder diesen mit `cbsScheduler.rescheduleEvent.fire()` asynchron aufrufen.

4.4.4.4 Einhaltung der CBS-Regeln

Mit den oben beschriebenen Punkten wurden zunächst die Stellen innerhalb eines CBS-basierten Schedulers in Betracht gezogen, an denen eine Aktualisierung einer Server-Deadline (CBS-Regeln 4 und 7) und eine Zustandsänderung eines Servers zwischen „aktiv“ und „inaktiv“ (CBS-Regeln

5, 7 und 8) stattfindet (vgl. Seite 89). Die restlichen Regeln werden durch die hier etablierte Lösung ebenfalls erfüllt:

1. Das Budget und die Periode des Servers wurden durch die gleichen Parameter eines zugrundeliegenden PGP-Objekts realisiert. Eine absolute Server-Frist wurde künstlich definiert und wird an entsprechenden Punkten CBS-konform aktualisiert.
2. Jedem Task-Job wird dynamisch die aktuelle Server-Deadline zugewiesen – dies wird durch die Aktivierung der Server mit ihren absoluten Deadlines realisiert.
3. Die aktuelle Server-Kapazität wird auf der PGP-Ebene automatisch durch die RT-JVM überwacht und mit den Kosten der gekapselten Tasks dekrementiert.
6. Eintreffende Jobs innerhalb eines CBS werden nach einer präemptiven EDF-Disziplin durch das interne Task-Scheduling innerhalb der Prioritäts-Warteschlangen des RTSJ-`PrioritySchedulers` verwaltet.
9. Zu jedem Zeitpunkt wird einem Job die letzte generierte Server-Deadline zugewiesen – dies wird ebenfalls durch die Aktivierung der Server mit ihren absoluten Deadlines realisiert.

4.4.5 Optimierung und Rekonfiguration von Applikationsmodi

Wie bereits in Kapitel 3.2.1 erklärt, löst die Aktualisierung des Sicherheitspuffers einer einzelnen Task eine Neuberechnung der Kapazität ihres zuständigen CBS aus. Dieser Vorgang propagiert bis zur obersten Ebene des Scheduling-Modells, wo eine Optimierung der Qualität der Anwendungen durch eine Neukonfiguration ihrer Modi vollzogen wird. Listing 4.17 zeigt die grobe Implementierung einer Optimierungs-Instanz.

Die eigentliche Optimierung wird innerhalb einer „nativen“ Methode `findSetOfModes()` bereitgestellt, die den im Kapitel 3.2.3 vorgestellten Rucksack-basierten Algorithmus in C implementiert und über das Java Native Interface (JNI) aufgerufen wird. Die native Implementierung des Algorithmus ist in [127] einsehbar¹⁴.

Als Eingabe für die Berechnung werden die ermittelten Bandbreiten und die Nutzen (Utilities) aller Applikationsmodi, sowie die aktuellen Prioritäten der Anwendungen bereitgestellt (vgl. Listings 4.2 und 4.3). Das Ergebnis ist eine Liste der optimalen Zielmodi, in welche die Anwendungen umgeschaltet werden müssen. Diese Berechnung wird nur dann durchgeführt, wenn die

¹⁴siehe <https://github.com/vladnikolov/arema/blob/master/src/framework/jni/optimization.c>

Summe der CBS-Bandbreiten der aktuellen Modi-Konfiguration der Anwendungen die verfügbare System-Kapazität R übersteigt (`checkOverload()`), oder wenn genügend freie Ressourcen für einen potentiell höheren Modus einer Anwendung verfügbar sind (`checkSpareCapacity()`).

```
1 public class Optimization {
2     Application[] apps;
3     AsyncEvent optimizationEvent;
4
5     int R = /* 128, 256, ... */
6
7     AsyncEventHandler optimizationHandler =
8         new AsyncEventHandler() {
9             public void handleAsyncEvent() {
10                 checkConfig();
11             }
12         }
13
14     native int[] findSetOfModes(...);
15
16     public void checkConfig() {
17         int[][] modes, utils;
18         int[] prios;
19
20         if (checkOverload() || checkSpareCapacity()) {
21             for (int i=0; i < apps.length; i++) {
22                 modes[i] = app.getBandwidths();
23                 utils[i] = app.getUtils();
24                 prios[i] = app.getPriority();
25             }
26
27             int[] res = findSetOfModes(R, modes, utilities, priorities);
28
29             for (int i=0; i < res.length; i++) {
30                 app[i].switchMode(res[i])
31             }
32         }
33     }
34 }
```

Listing 4.17: Optimierung

Der Integer-Wert von R definiert außerdem die Ressourcenauflösung im System. Wie bereits in Kapitel 3.2.3 erklärt, hat diese Auflösung entscheidende Auswirkungen auf die Laufzeit des Optimierungsalgorithmus und kann im Voraus auf einen beliebigen Wert eingestellt werden. Die übergebenen Bandbreiten der Modi müssen während der Optimierung innerhalb dieser Metrik konvertiert werden, was automatisch in `getBandwidths()` passiert.

In Listing 4.17 wird die Aktivierung der Zielmodi der Anwendungen konsekutiv mit steigendem Applikations-Index durchgeführt. Andere Strategien sind jedoch ebenfalls möglich, wie zum Bei-

spiel eine, die Applikationsmodi zunächst herunterschaltet und somit freie Ressourcen für höhergestufte Anwendungen verschafft. Während des Umschaltungsprozesses werden Aktualisierungen der Sicherheitspuffer der Tasks temporär deaktiviert, um potentiell rekursive Rekonfigurationen zu unterbinden.

Die Optimierung der Applikationsmodi kann ähnlich wie das Rescheduling der CBS durch ein asynchrones Ereignis `optimizationEvent` ausgelöst werden. Die eigentliche Prozedur wird durch einen `optimizationHandler` ausgeführt, der mit einer höheren Priorität als die der Rescheduling-Handler der CBS eingestellt ist, um Rekonfigurations-Verzögerungen durch das CBS- und Task-Scheduling zu vermeiden. Dies ist notwendig, um beim Auftreten von Fristverletzungen eine sofortige Gegenmaßnahme einleiten zu können.

Da Optimierungen zunächst als Folge von Aktualisierungen der Kostenschätzungen von Tasks auftreten können, wird der Aufwand der Optimierungs-Handler in die Approximationsvorgänge der Task-Kosten mit berechnet (siehe Kapitel 4.4.7). Darüber hinaus können Optimierungen durch Benutzervorgänge, wie die Priorisierung und die Installation bzw. Deinstallation von Anwendungen, ausgelöst werden. Da letztere nicht vorhersehbar sind, werden sie durch spezielle Schnittstellen und Aktivitätsträger im Framework unterstützt und nur optimistisch mit freien Kapazitäten im System bedient.

4.4.6 Implikationen der Umschaltungen von Anwendungsmodi

Wie bereits in den Kapiteln 4.4.1 und 4.4.2 erklärt, propagiert der Modus-Wechsel einer Anwendung mit `Application.switchMode(ApplicationMode)` bis zur Task-Ebene des hierarchischen Modells. Dabei werden die Applikations-Tasks durch `Task.activateMode(TaskMode)` in entsprechenden Zielmodi für den jeweiligen Anwendungsmodus aktiviert (siehe Listings 4.3 und 4.4). Darüber hinaus wird der CBS der umgeschalteten Applikation mit der benötigten Kapazität und Periode für den neuen Anwendungsmodus rekonfiguriert (siehe Listing 4.9). Bei der Umschaltung der Task-Modi werden auch taskmodus-spezifische `ReleaseParameters` gesetzt, womit die Perioden und die Deadlines der Tasks ihren Zielmodi entsprechend angepasst werden. Nach der RTSJ-Spezifikation greifen die Perioden- und Deadline-Änderungen von Echtzeit-Threads erst ab ihrer nächsten Aktivierungs-Periode. Dasselbe gilt auch für Veränderungen der Parameter, wie Kapazitäten und Perioden, der den CBS zugrunde liegenden PGP-Objekte [27].

Wie bereits in Kapitel 3.2.2 erklärt, werden die CBS-Perioden mit $P_i = \min(T_{i,k})$ gleich den kleinsten Task-Perioden ihrer Anwendungen in ihren jeweiligen Modus gesetzt. Aus diesem

Grund ist es beim Modus-Wechsel einer Anwendung prinzipiell möglich, dass die Rekonfiguration ihres CBS innerhalb der RT-JVM schon greift, aber manche Tasks mit Perioden größer als $P_{i,j}$ noch nicht in ihren neuen Zielmodi ausführen. Die Konsequenz ist, dass Tasks durch die noch laufenden Jobs ihrer alten Modi bereits die Server-Kapazität für den neuen Anwendungsmodus verbrauchen können. Dies kann zu einer vorzeitigen Erschöpfung der Kapazität und Verdrängung eines Servers und somit zu einer Störung des Scheduling auf der Applikations-Ebene führen, deren Folgen sich als Fristverletzungen auf der Task-Ebene zeigen können.

```
1 public class ApplicationMode {
2     ...
3     public void activate() {
4         for (Task task : taskModes.keySet()) {
5             task.activateMode(taskModes.get(task));
6         }
7
8         app.cbs.reconfigure();
9         app.cbs.suspend()
10    }
11 }
12
13 public class CBS extends ProcessingGroupParameters {
14     ...
15     AbsoluteTime absoluteDeadline;
16     boolean active;
17
18     public void suspend () {
19         setCostOverrunHandler(null);
20         setRemainingBudget(new RelativeTime(0,0));
21         absoluteDeadline = 0;
22         active = false;
23     }
24     ...
25 }
26
27 public class Task extends RealtimeThread {
28     ...
29     void leaveBarrier() {
30         ...
31         if (!cbs.active) {
32             ...
33             if (cbs.getCostOverrunHandler == null) {
34                 cbs.setCostOverrunHandler(overrunHandler)
35             }
36         }
37         ...
38     }
39     ...
40 }
```

Listing 4.18: Code-Änderungen für einen Applikationsmodus-Wechsel

Um diese Charakteristik zu umgehen, wurde beim Wechsel eines Task-Modus die Möglichkeit etabliert, einen gerade laufenden Job durch das Auslösen einer AIE `jobAIE` abzuberechnen (siehe Listing 4.5, Zeile 9). Auf diese Weise wird verhindert, dass Tasks bis zum Wechsel ihrer modus-spezifischen Perioden, Deadlines und Logik weitere Kosten erzeugen und Server-Kapazität verbrauchen. Für eine Bandbreiten-konsistente Aktivierung eines Servers nach einem Modus-Wechsel, wird dieser nach Abbruch der Task-Jobs mit einer Kapazität von $c_S = 0$ und einer Deadline $D_S = 0$ suspendiert. Somit wird bei der ersten Aktivierung einer Task im neuen Anwendungsmodus, die Server-Kapazität mit dem neuen Wert für $R_{i,j}$ aufgeladen und eine entsprechende Server-Deadline gesetzt (CBS-Regel 7). Die dafür notwendigen Code-Änderungen werden in Listing 4.18 zusammengefasst.

Am Ende von `ApplicationMode.activate()` wird der Anwendungs-Server suspendiert (Zeile 9). Bei der Implementierung von `CBS.suspend()` wird wieder die nicht RTSJ-konforme Funktion `setRemainingBudget()` benötigt (Zeile 20), um die restliche Server-Kapazität c_{S_i} gleich Null zu setzen (die Implementierung dieser Funktion wird später im Kapitel 4.5 vorgestellt). Davor muss der Overrun-Handler des CBS kurzfristig deaktiviert werden, um eine Deadline-Verlängerung aufgrund der Kapazitäts-Entleerung zu verhindern. Der Overrun-Handler wird dann in `Task.leaveBarrier()`, bei der Aktivierung der ersten Task im neuen Anwendungsmodus, erneut zusammen mit der neuen Server-Deadline und -Kapazität gesetzt.

Der genaue Vorgang beim Umschalten eines Anwendungsmodus ist nun durch die folgenden Schritte auf der Code-Ebene implementiert:

- Durch `Application.switchMode(ApplicationMode)` wird ein neuer Anwendungsmodus aktiviert (Listing 4.2).
- In `ApplicationMode.activate()` (Listing 4.18) werden alle Anwendungs-Tasks in entsprechende Zielmodi umgeschaltet. Der Server wird mit der Periode $P_{i,j}$ und der maximalen Kapazität $R_{i,j}$ für den neuen Anwendungsmodus rekonfiguriert und mit einer Rest-Kapazität $c_{S_i} = 0$ und Deadline $D_{S_i} = 0$ suspendiert. Der Overrun-Handler des CBS wird temporär deaktiviert.
- In `Task.activateMode(TaskMode)` (Listing 4.5) wird der aktuelle Job einer jeden Task zunächst abgebrochen. Der alte Task-Modus wird deaktiviert und der neue entsprechend aktiviert. Zum Schluss werden die Definitionen der Task-Periode und -Deadline durch zielmodus-spezifische `ReleaseParameters` ausgetauscht. Diese gelten ab der nächsten Task-Aktivierung.

- Bei der ersten Aktivierung einer Task mit ihrer neuen Periode und Frist wird in `Task.leaveBarrier()` (Listing 4.14) der Server aktiviert. Dabei wird nach der Aktivierungsregel 7 die Server-Kapazität mit dem neuen maximalen Wert $c_{S_i} = R_{i,j}$ aufgeladen und, ausgehend von einem Aktivierungs-Zeitpunkt t , eine neue Server-Deadline $D_{S_i} = t + P_{i,j}$ gesetzt. Darüber hinaus wird der Overrun-Handler wieder gesetzt (Listing 4.18) und der `CBSScheduler` aufgerufen. Dabei kann der CBS temporär verdrängt werden.
- Bei der weiteren Ausführung der Task in `wFNP()` und `run()` (siehe Code-Listing 4.5) wird die neue, modus-spezifische Logik durch `jobAIE.doInterruptible(activeMode)` ausgeführt, dabei verbraucht die Task die zuvor eingestellte (neue) Kapazität des Servers.
- Die restlichen Anwendungs-Tasks werden entsprechend den Phasen ihrer letzten Jobs aktiviert, während der Server bereits aktiv ist. Hier greift nun das CBS-Scheduling, dessen Implementierung in Kapitel 4.4.4 beschrieben wurde. Da die Planbarkeitsanalyse für die Kapazitäts-Bemessung des Zielmodus für einen synchronen Task-Start (worst-case) berechnet wurde, muss die Aktivierung der Tasks innerhalb des neu dimensionierten Servers, unabhängig von ihren Aktivierungs-Phasen, ab dem Rekonfigurationszeitpunkt erfolgreich, ohne Fristverletzungen ablaufen.

4.4.7 Überprüfung und Aktualisierung der Task-Approximationen

Im Folgenden wird die Implementierung der Kosten-Approximation auf der Task-Ebene vorgestellt. Diese wird durch die Sicherheitspuffer der Tasks realisiert, die dynamisch überprüft und bei Bedarf neu eingestellt werden müssen. Die hierfür notwendigen Code-Erweiterungen auf der Task-Ebene sind zunächst in Listing 4.19 aufgeführt.

Die Approximationen der Task-Kosten sind modus-spezifisch und werden anhand der periodisch gemessenen Ausführungszeiten der Tasks berechnet. Dafür wird die Definition eines `TaskMode` aus Listing 4.4 zunächst mit Gleitkomma-Variablen für den aktuellen Mittelwert `meanCosts`, für die aktuelle Standardabweichung `sigma` und für die obere `sbTop` und untere `sbBottom` Grenze des Sicherheitspuffers einer Task erweitert. Die Berechnung findet innerhalb der Methode `resSample()` statt und wird automatisch am Ende eines jeden periodischen Task-Jobs ausgeführt (vgl. Listing 4.5).

```
1 public abstract class TaskMode implements Interruptible {
2     ...
3     double sbTop, sbBottom, meanCosts, sigma;
4     static final double ALPHA = 0.125;
5     boolean rateBasedSafetyBufferCheck = true;
6     Optimization optimization;
7     Queue<Double> samples;
8     CBS cbs;
9
10    void resSample (double measuredValue) {
11        if(!samples.offer(measuredValue)) {
12            samples.poll();
13            samples.offer(measuredValue);
14        }
15
16        meanCosts = (1 - ALPHA) * meanCosts + ALPHA * measuredValue;
17
18        if (!rateBasedSafetyBufferCheck)
19            if (checkSafetyBuffer()) {
20                updateSafetyBuffer();
21                cbs.updateCbsCapacity();
22                optimization.checkConfig();
23            }
24    }
25
26    boolean checkSafetyBuffer () {
27        return ((meanCosts < sbBottom)|| (sbTop < meanCosts))
28    }
29
30    updateSafetyBuffer () {
31        sigma = computeStandardDeviation();
32        sbTop = meanCosts + 2 * sigma;
33        sbBottom = meanCosts - 2 * sigma;
34    }
35
36    double getEstimate () { return sbTop; }
37 }
```

Listing 4.19: Erweiterung der Task-Modus-Implementierung

In den Zeilen 11 – 14 wird zunächst ein gleitendes Fenster über die gemessene Zeitreihe durch eine Warteschlange implementiert, das für die Berechnung der Standardabweichung benötigt wird. Die Größe des Fensters wird bei der Instanziierung der Warteschlange durch `samples = new ArrayBlockingQueue<Double>(1024)` definiert, kann jedoch auch zur Laufzeit verändert werden. Die Fenstergröße spielt eine wichtige Rolle bei der automatischen Erfassung von periodischen Task-Bursts, wie später im Kapitel 5.5 gezeigt wird. In Zeile 16 wird der gleitende Mittelwert der gemessenen Task-Kosten mit einem bestimmten Glättungsfaktor `ALPHA` berech-

net. Dieser Faktor ist in der aktuellen Implementierung auf den Wert `ALPHA= 0.125` eingestellt, er kann jedoch auch dynamisch verändert werden. Mit dieser Voreinstellung¹⁵ wird zunächst ein gegenüber kleineren Schwankungen relativ stabiler Mittelwert implementiert. Er besitzt jedoch eine träge Anpassungsgeschwindigkeit an stärkere Veränderungen der Task-Kosten.

Wie bereits in Kapitel 3.2.1 erklärt, werden die Sicherheitspuffer der Tasks *ratenbasiert* oder *instantan* überprüft und gegebenenfalls aktualisiert. Die Strategie kann für jede Task dynamisch entschieden und separat eingestellt werden, je nachdem ob diese gerade unter Fristverletzungen leidet oder nicht.

In Listing 4.19 ist zunächst die instantane Überprüfung aufgeführt, die auf der Task-Ebene innerhalb einer jeden Task-Periode stattfindet. Hierbei werden in `checkSafetyBuffer()` (Zeile 19) die mittleren Task-Kosten mit den aktuellen Grenzen des Sicherheitspuffers verglichen. Wenn die Grenzen verletzt werden, dann wird der Sicherheitspuffer in `updateSafetyBuffer()` mit der aktuellen Standardabweichung `sigma` aktualisiert (die Implementierung der Methode `computeStandardDeviation()` ist in [127] ersichtlich¹⁶). Anhand dessen werden daraufhin die Grenzen des Sicherheitspuffers neu eingestellt. Somit ergibt sich durch die obere Grenze des Puffers (vgl. Zeile 36) eine neue Kostenschätzung der Task. Mit der neuen Schätzung wird in `updateCbsCapacity()` anhand einer Planbarkeitsanalyse eine neue CBS-Kapazität und Bandbreite für die gesamte Anwendung ermittelt (vgl. Kapitel 3.2.2). Die Implementierung dieser Methode kann ebenfalls in [127] eingesehen werden¹⁷. Die neu errechnete Kapazität gilt für den aktuellen Anwendungsmodus ab der nächsten Kapazitätsaufladung. Aufgrund der Bandbreiten-Änderung wird die Optimalität der Modi-Konfiguration der Anwendungen in `optimization.checkConfig()` überprüft. Bei einer instantanen Überprüfung werden alle notwendigen Berechnungen und Aktualisierungs-Vorgänge durch die jeweilige Task selbst durchgeführt.

Für eine ratenbasierte Überprüfung und Aktualisierung der Sicherheitspuffer, kann prinzipiell eine beliebige Periode ausgewählt werden. Hier ist ein Trade-off zwischen einer niedrigeren Rekonfigurationsrate und der Gefahr von Fristverletzungen notwendig (vgl. Kapitel 3.2.1), „Periodische Überprüfung der Sicherheitspuffer“. Da die benötigte Server-Kapazität einer Anwendung für die Hyperperiode ihrer Tasks ermittelt wird, innerhalb welcher sich die applikations-lokale Task-Aktivierung wiederholt, und Veränderungen der Kostenschätzungen durch die Sicherheitspuffer möglichst verzögert werden, ist die Hyperperiode der Tasks einer Anwendung ein geeigneter Zeitpunkt für die ratenbasierte Überprüfung der Sicherheitspuffer. Die Überprüfung wurde mit

¹⁵hierbei findet eine stärkere Gewichtung der gemessenen Historie

¹⁶<https://github.com/vladnikolov/arema/blob/master/src/framework/src/arema/framework/task/TaskMode.java>

¹⁷<https://github.com/vladnikolov/arema/blob/master/src/framework/src/arema/framework/cbs/CBS.java>

einem RTSJ `PeriodicTimer` auf der CBS-Ebene realisiert. Die hierfür notwendigen Erweiterungen der CBS-Definition aus Listing 4.9 werden im folgenden Listing 4.20 dargestellt:

```

1 public class CBS extends ProcessingGroupParameters {
2     ...
3     PeriodicTimer periodTimer;
4
5     AsyncEventHandler periodicTimerHandler = new AsyncEventHandler() {
6         public void handleAsyncEvent() {
7             boolean recomputeBandwidth = false;
8             for (Task task : app.getTasks())
9                 if(task.activeMode.rateBasedSafetyBufferCheck)
10                    if (task.activeMode.checkSafetyBuffer()) {
11                        task.activeMode.updateSafetyBuffer();
12                        recomputeBandwidth = true;
13                    }
14
15                if (recomputeBandwidth) {
16                    updateCbsCapacity();
17                    recomputeBandwidth = false;
18                    optimization.checkConfig();
19                } } }
20
21     public void reconfigure() {
22         ...
23         RelativeTime period = app.activeMode.computeTasksHyperperiod();
24         if(periodTimer.isRunning()) periodTimer.destroy();
25         periodTimer = new PeriodicTimer(null, period, null,
26             periodicTimerHandler).start();
27     }
28     ...
29 }

```

Listing 4.20: Constant Bandwidth Server-Implementierung (4)

Der `PeriodicTimer` einer Anwendung wird im Zuge eines Modus-Wechsels in `reconfigure()` mit der aktuellen Hyperperiode ihrer Tasks eingestellt und gestartet. Die entsprechenden periodischen Ereignisse des Timers werden innerhalb eines `periodicTimerHandlers` behandelt. Dabei werden die Sicherheitspuffer aller Tasks, ähnlich wie bei der instantanen Strategie, überprüft und notfalls aktualisiert. Falls es zu einer Pufferaktualisierung kommt, wird zum Schluss die Server-Kapazität aktualisiert und die Optimierung der Modi (synchron) aufgerufen.

Planung der Puffer-Aktualisierungen

Bei einer instantanen Überprüfung der Sicherheitspuffer (siehe Listing 4.19) führen die Tasks die potentiellen Aktualisierungsvorgänge in `updateSafetyBuffer()`, die applikations-lokale Plan-

barkeitsanalyse in `cbs.updateCbsCapacity()` und die Optimierung der Anwendungsmodi in `optimization.checkConfig()` selbst durch. Der dabei anfallende Aufwand wird somit den Tasks selbst zugetragen und innerhalb der Task-Kosten-Messungen berücksichtigt. Dies kann zu einer (kurzfristigen) Erhöhung ihrer Kosten führen. Es ist unwahrscheinlich, dass eine Task nach einer Aktualisierung ihres Puffers diesen sofort wieder verletzen wird. Der zusätzliche Aufwand wird somit automatisch als Jitter in die laufende Kostenschätzung der jeweiligen Task aufgenommen und dadurch im System eingeplant. Damit verbundene, potentielle Ausbrüche der Task aus ihrem aktuellen Konfidenzintervall, werden automatisch durch einen neuen Approximationspuffer und Schätzung ihrer Kosten abgedeckt und mit der erforderlichen Güte im System unterstützt.

Der Aufwand für die ratenbasierte Aktualisierung der Sicherheitspuffer und die Optimierung der Modi innerhalb der `PeriodicTimer`-Handler der Anwendungen kann ebenfalls im System beschränkt und eingeplant werden. Da die Periode T_{H_i} mit der die Handler aktiviert werden bekannt ist und pro Anwendung ein solcher Handler existiert, kann über eine Planbarkeitsanalyse eine globale Kapazität für die Überprüfung der Sicherheitspuffer alloziert und von der Menge der Gesamtressourcen R abgezogen werden. Dabei werden anhand der „schlimmsten“ gemessenen Ausführungszeiten der Handler dynamisch obere Schranken C_{H_i} ihrer Kosten ermittelt. Diese setzen sich aus den einzelnen Kosten für die Berechnung der Standardabweichung und die Aktualisierung der Task-Puffer (C_{SB_i}), für die applikations-lokale Planbarkeitsanalyse (C_{FA_i}) und für die Optimierung und Umschaltung der Anwendungsmodi (C_{OPT_i}) zusammen.

$$C_{H_i} = C_{SB_i} + C_{FA_i} + C_{OPT_i} \quad (4.1)$$

Für eine garantierte Bereitstellung von Ressourcen im Falle einer potentiellen Pufferaktualisierung müssen die Handler-Kosten periodisch eingeplant werden. Da die Ereignis-Behandlung keiner Frist unterzogen ist, kann hierfür durch $\sum_i \frac{C_{H_i}}{T_{H_i}} = B_H$ eine benötigte Bandbreite B_H ermittelt werden und von R subtrahiert werden. Somit ist $R = R - B_H$.

Die Berechnung der Standardabweichung der gemessenen Kosten einer Task und die Aktualisierung ihrer Puffer-Werte ist mit einem konstanten Aufwand verbunden. Aus diesem Grund kann C_{SB} durch eine Konstante für den schlimmsten Fall repräsentiert werden, bei dem die Puffer aller Anwendungs-Tasks aktualisiert werden müssen (siehe Listing 4.20, Zeilen 10 und 11). Der Wert von C_{FA_i} ist abhängig von der Anzahl der Punkte innerhalb der Task-Hyperperiode einer Anwendung, an denen die Planbarkeitsanalyse in `updateSafetyBuffer()` die Rechenzeitanforderung der Tasks überprüfen muss (siehe Kapitel 3.2.2). Da die Approximationsschranke t' abhängig von den partiellen Taskauslastungen ist, kann dieser Wert variieren, jedoch auch

für den schlimmsten Fall der ganzen Hyperperiode beschränkt werden¹⁸. Der Wert von C_{OPT_i} teilt sich zunächst in die tatsächlichen Kosten für die Modi-Optimierung und die Kosten für die Umschaltung der Anwendungsmodi auf. Der Aufwand für die Optimierung, bei unveränderter Konfiguration der Anwendungen in einem System, bleibt jedoch soweit konstant (siehe Kapitel 3.2.3). Lediglich die modus-abhängigen Task-Rekonfigurationen (in `activate()` bzw. `deactivate()` in Listing 4.4) können für weitere Schwankungen sorgen. In ARTOS werden zur Laufzeit anhand der gemessenen Werte obere Schranken für die C_{H_i} ermittelt. Eine Aktualisierung der reservierten Bandbreite B_H für die Ereignis-Behandlung wird nur dann gemacht, wenn ein neuer „schlimmster“ Wert von einem C_{H_i} ermittelt wurde.

4.5 Bewertung und Erweiterungen von RTSJ

Im Folgenden werden bestimmte problematische Aspekte näher diskutiert, welche speziell aufgrund der Implementierung des Modells mit RTSJ zum Tragen kamen. Dazu gehören Effekte, die bei Änderungen von Task-Perioden und -Fristen entstehen können, benutzerdefinierte Kapazitätssteuerung für Prozess-Gruppen, sowie grundlegende Überlegungen zur Implementierung benutzerdefinierter Scheduling-Mechanismen innerhalb einer RT-JVM. Viele der besprochenen Probleme treten aufgrund von Mängeln innerhalb der Spezifikation auf, entweder wegen grundsätzlich fehlender Mechanismen oder aufgrund fehlender Schnittstellen zu Funktionen, die von einer RT-JVM ohnehin bereitgestellt werden müssen. Die in diesem Kapitel vorgestellten Erweiterungen und Anpassungen der RTSJ Schnittstellen und der bereitgestellten Funktionalität einer RT-JVM, wurden bereits in [100] publiziert und im Rahmen der RTSJ Expert Group diskutiert. Einige der besprochenen Vorschläge sind bereits in die neue Version 2.0 der Spezifikation eingeflossen, welche in Kürze veröffentlicht wird.

4.5.1 Änderungen von Task-Perioden und -Fristen

Wie zuvor in Kapitel 4.4.3 erklärt, wurde das applikations-lokale EDF-Scheduling auf der Task-Ebene über den `RTSJ-PriorityScheduler` anhand einer für jede Task explizit verwalteten, absoluten Frist `absoluteDeadline` realisiert. Diese entspricht generell dem Zeitpunkt, an dem die relative Frist einer Task innerhalb ihrer laufenden Periode fällig wird. Dieser Zeitpunkt ist per Definition äquivalent zu $r_{n_{i,k}} + D_{n_{i,j,k}}$, wobei $r_{n_{i,k}}$ den Zeitpunkt der n -ten periodischen Aktivierung einer Task $\tau_{i,k}$ und $D_{n_{i,j,k}}$ ihre aktuelle modus-spezifische, relative Frist angibt. In

¹⁸in diesem Fall müssen alle Deadline-Punkte innerhalb der Task-Hyperperiode berücksichtigt werden

der Funktion `Task.leaveBarrier()` (in Listing 4.5) wird $r_{n_{i,k}}$ durch eine Variable `periodBegin` repräsentiert, die zu Beginn einer jeden Task-Periode inkrementiert und zur Berechnung der aktuellen absoluten Task-Deadline verwendet wird. Unter der Voraussetzung, dass die exakte Startzeit einer jeden Task bekannt ist, ist diese Implementierung nahezu ausreichend. In der Praxis ist jedoch die Determination der exakten absoluten Deadline einer Task schwieriger, da (a) ihre Perioden-Länge und relative Frist sich durch Modi-Wechsel zu beliebigen Zeitpunkten verändern können und (b) die Task in ihrem Null-Modus deaktiviert und zu einem unbestimmten Zeitpunkt wieder reaktiviert werden kann (vgl. Kapitel 4.4.3). Die Implementierung mit der JamaicaVM erfordert hierbei besondere Vorsicht, da beide Situationen zu falschen Werten für `periodBegin` und `absoluteDeadline` führen können. Dadurch kann eine Inkonsistenz zwischen dem Scheduler auf der Task-Ebene des hierarchischen Modells und das effektive Verhalten der Tasks innerhalb des unterliegenden `PrioritySchedulers` der RT-JVM entstehen.

Um eine kohärente Sicht beider Scheduler zu etablieren, wird ein Mechanismus benötigt, mit dem die exakte absolute Zeit $r_{n_{i,k}}$ der aktuellen periodischen Aktivierung eines Echtzeit-Threads programmatisch bestimmt werden kann. Damit könnte die absolute Frist einer Task direkt nach ihrer Deblockierung von `waitForNextPeriod()` auf $r_{n_{i,k}} + D_{n_{i,j,k}}$ gesetzt werden. Leider bietet RTSJ keinen standardmäßigen Mechanismus dafür an. Bei der Implementierung des Modells war es jedoch möglich, über Java-Reflection auf einen internen Parameter der JamaicaVM `RealtimeThread.nextPeriod` vom Typ `AbsoluteTime` zuzugreifen, der einen vergleichbaren Zweck erfüllt. Anhand dessen kann der Wert von `absoluteDeadline` periodisch auf den richtigen Zeitpunkt gesetzt werden. Nichtsdestotrotz, hängt die korrekte Behandlung von Perioden- und Fristen-Änderungen anhand von `nextPeriod` davon ab, ob eine Modifikation ihrer Werte aufgetreten ist, bevor oder nachdem ein Echtzeit-Thread bereits in `waitForNextPeriod()` suspendiert wurde.

In Bezug auf die Periode eines Echtzeit-Threads schreibt die aktuelle RTSJ Spezifikation (1.0.2) Folgendes vor:

“Subsequent release events occur as each period falls due, at times determined as follows: if the i^{th} release event occurred at a time t_i , then the $i + 1$ release event occurs at the time $t_i + T_i$, where T_i is the value of the period stored in the real-time thread’s `PeriodicParameters` object at the time t_i .”[27]

Dementsprechend hat allein die Periode T_i zum Zeitpunkt t_i eine Auswirkung auf den nächsten Aktivierungspunkt t_{i+1} , unabhängig davon wie T_i bis dahin verändert werden könnte.

Im Gegensatz dazu ist das Verhalten der JamaicaVM leicht abgewandelt. Der Parameter `nextPeriod` wird in `waitForNextPeriod()` am Ende einer jeden periodischen Aktivierung, an-

hand der aktuellen Thread-Periode, aktualisiert. Also repräsentiert `nextPeriod` den absoluten Zeitpunkt t_{i+1} , an dem der Thread für seine $(i+1)^{th}$ -Periode deblockiert wird. Die Aktivierungszeit t_{i+1} wird jedoch nicht zum Zeitpunkt t_i mit der aktuellen Periode T_i innerhalb der `PeriodicParameters` des Threads gesetzt, sondern mit der Periode zum Zeitpunkt seiner Suspendierung t_{wfnp} . Wenn nun ein Perioden-Wechsel zwischen t_i und t_{wfnp} stattfindet, hat dieser einen sofortigen Effekt auf die Länge der aktuellen Periode, d.h. auf die Blockierungsdauer des Threads in `waitForNextPeriod()` und auf seinen nächsten Aktivierungszeitpunkt t_{i+1} . Andererseits, wenn ein Wechsel, während ein Thread in `waitForNextPeriod()` blockiert ist, stattfindet, entspricht das Aktivierungs-Verhalten der RTSJ Spezifikation.

In einigen Fällen ist das beschriebene Verhalten der JamaicaVM gegenüber der Spezifikation von Vorteil, da es eine schnellere Adaption an eine neue Aktivierungs-Frequenz einer Task ermöglicht. Speziell wenn zum Beispiel eine lange Periode bis zum Schluss abgewartet werden muss, kann eine sofortiger Wechsel zu einer kürzeren Periode die Reaktionsfähigkeit eines Systems verbessern. Im Allgemeinen ist eine Perioden-Vergrößerung unproblematisch, da sie lediglich eine Verlängerung des nächsten Aktivierungs-Zeitpunkts $(i+1)$ eines Threads impliziert. Eine Perioden-Verkürzung kann dagegen zu n konsekutiven Aktivierungen des Threads ohne Blockierung in `waitForNextPeriod()` führen. Dieser Fall ist in Abbildung 4.6 beispielhaft dargestellt.

Abbildung 4.6: Perioden-Verkürzung mit JamaicaVM (aus [100])

Angenommen, ein periodischer Echtzeit-Thread τ_1 wird zum Zeitpunkt t_i mit einer Periode T_1 aktiviert und vollständig bis zu seiner Suspendierung in `waitForNextPeriod()` ausgeführt. An einem beliebigen Zeitpunkt dazwischen wird seine Periode durch ein externes Ereignis auf einen kleineren Wert gesetzt, z.B. auf $1/8$ seiner alten Periode. Wenn der Thread `waitForNextPeriod()` betritt, dann wird der Zeitpunkt seiner nächsten Aktivierung durch `nextPeriod =  $t_i + T_2$` bestimmt, was allerdings in diesem Fall einem Wert in der Vergangenheit entspricht. Als Konsequenz daraus blockiert `waitForNextPeriod()` solange nicht, bis `nextPeriod` die aktuelle Zeit t eingeholt hat. In Abbildung 4.6 wird dies durch die gestaffelten

Jobs des Threads verdeutlicht. Wieviele logisch *ausstehenden* Aktivierungen in einem solchen Fall erzeugt werden, hängt von der Länge der kleineren Periode T_2 und von der Zeitspanne zwischen t_1 und dem Eintritt in `waitForNextPeriod()` ab, d.h. $n_{pending} = \lfloor (t_{wnp} - t_i) / T_2 \rfloor$.

Eine konsekutive Ausführung von ausstehenden Perioden ohne Blockieren entspricht effektiv nicht der vorgegebenen Aktivierungsfrequenz einer Task. Es ist generell fraglich, welche Arbeit von einer Task während solcher künstlich erzeugten Aktivierungen unzeitgemäß verrichtet werden kann. Aus diesem Grund wurde für diesen Fall eine Strategie gewählt, bei der die Job-Ausführung für $n_{pending}$ Perioden ausgesetzt wird. Dies beschleunigt den Prozess bis `nextPeriod` und `absoluteDeadline` sich wieder in einem konsistenten Zustand befinden. Wann immer ein Task-Modus Wechsel eine Perioden-Verkleinerung impliziert, wird das Auftreten von ausstehenden Aktivierungen überprüft und das Verhalten wie beschrieben angepasst.

Eine ähnliche Bedingung existiert auch für Task-Fristen. Für diese schreibt RTSJ in analoger Weise vor, dass die Zeit der i -ten Frist einer Task abhängig ist vom Wert ihres Deadline-Parameters D_i zum Aktivierungszeitpunkt t_i [27]. Leider war es nicht möglich eine JVM-interne Representation der aktuellen i -ten Frist eines Echtzeit-Threads zu extrahieren. Aus diesem Grund wurde auf der Task-Ebene ein künstlicher Parameter `absoluteDeadline` bereitgestellt. In der JamaicaVM wird die i -te Deadline einer Task in einer ähnlichen Weise während des letzten Eintritts in `waitForNextPeriod()` bestimmt und nicht beim Aktivierungszeitpunkt t_i . Demzufolge, ist die effektive Frist einer Task innerhalb ihrer nächsten Periode nicht up-to-date, wenn eine Änderung ihrer relativen Frist auftritt, während sie bereits in `waitForNextPeriod()` blockiert ist und auf ihre nächste Aktivierung wartet. Wenn die relative Frist bei einem Modus-Wechsel verkürzt wird, bleibt somit ihre effektive JVM-interne Ablaufzeit für eine weitere Periode lang. Umgekehrt, wenn die Frist im Zuge eines Modus-Wechsels vergrößert wird, kann das Scheduling auf der Task-Ebene eine unerwartete Fristverletzung hervorrufen, da die Task-Logik bereits gewechselt wurde, jedoch die effektive Ablaufzeit der Frist kurz bleibt. Für den letzten Fall wird erneut die Job-Ausführung für eine Task-Periode ausgesetzt, bis `absoluteDeadline` und der Ablauf der i -ten relativen Frist der Task synchron sind.

Zusammenfassend lässt sich anmerken, dass es für die Implementierung von benutzerdefinierten Scheduling-Mechanismen auf der Basis von RTSJ notwendig ist, die exakte absolute Zeit der letzten Thread-Aktivierung zu kennen. RTSJ 2.0 [132] enthält eine entsprechende Methode innerhalb der Klasse `RealtimeThread`:

```
public AbsoluteTime getLastReleaseTime();
```

Anhand dieser Methode und einer RTSJ 2.0-konformen JVM kann die absolute Frist eines Threads innerhalb seiner aktuellen Periode bestimmt und somit ein benutzerdefiniertes Scheduling mit Standard RTSJ-Funktionen implementiert werden. Zum Implementierungszeitpunkt der vorgestellten Scheduling-Mechanismen innerhalb dieser Arbeit existiert jedoch noch keine Referenz-Implementierung der RTSJ 2.0 Spezifikation.

4.5.2 Prozess-Gruppen

Wie bereits im Kapitel 4.4.4 erwähnt, fehlen in der aktuellen RTSJ Spezifikation der `ProcessingGroupParameters` grundlegende Mechanismen, um diese als Basis für die Implementierung benutzerdefinierter Server zu verwenden. Einige bereits existierende Arbeiten, wie zum Beispiel [128, 129], schlagen ebenfalls Erweiterungen der PGPs vor, um Server-Implementierungen zu ermöglichen. Im Rahmen dieser Arbeit konnten CBS-ähnliche Spezialisierungen der PGPs, zur Kapselung von Applikationen mit mehreren Tasks, realisiert werden. Dabei wurde die RT-JVM-interne Kostenüberwachung der PGPs für Kapazitäts-Kontrolle der CBS verwendet. Für die Steuerung der Server wurde ein EDF-basierter Mechanismus, anhand Verschiebungen ganzer Anwendungen im RTSJ-Prioritätsraum, entwickelt. Die Bedingungen für die tatsächliche Aktivierung und Suspension der Server werden auf der Task-Ebene des hierarchischen Modells überprüft und entsprechende Scheduling-Ereignisse von dort aus initiiert (siehe Kapitel 4.4.4). Auf der anderen Seite mussten spezielle Mechanismen für ein benutzergesteuertes Aufladen der Kapazität eines Servers bereitgestellt werden, da die von den PGPs hierfür unterstützte Strategie nicht mit dem CBS-Algorithmus kompatibel ist. Im Allgemeinen waren zwei Anpassungen und Erweiterungen der RTSJ Funktionen notwendig:

1. Spezielle Methoden für das Abfragen und Setzen der aktuell verbleibenden Kapazität cs_i eines Anwendungs-Servers.
2. Unterbinden der periodischen Kapazitäts-Aufladung zu Beginn einer jeden PGP-Periode.

Die Lösung, welche im Rahmen dieser Arbeit vorgeschlagen und zusammen mit den aicas Entwicklern innerhalb der JamaicaVM implementiert wurde, wird im Folgenden genauer vorgestellt.

Die Kostenüberwachung für die PGPs wird innerhalb der JamaicaVM anhand von zwei internen Parametern `currentCost` und `remainingCost` umgesetzt. Der Parameter `currentCost` gibt die zu einem Zeitpunkt innerhalb der jeweiligen PGP-Periode verbrauchten kumulativen Kosten der

durch ein PGP-Objekt gekapselten Echtzeit-Threads an, während **remainingCost** die verbleibende Kapazität innerhalb der PGP-Periode beinhaltet. Die Überprüfung der verbrauchten Kosten innerhalb der JamaicaVM wird, ausgehend vom Beginn einer jeden PGP-Periode, etabliert. Der Mechanismus wird im Folgenden anhand des Pseudo-Codes in Algorithmus 3 erklärt.

Zu Beginn einer jeden Periode wird die verbleibende Kapazität **remaining** des PGP-Objekts auf seinen voreingestellten Wert **cost** gesetzt (aufgefüllt). Danach fährt die JVM eine Strategie zur Überprüfung der Thread-Kosten, bei der die Tatsache ausgenutzt wird, dass innerhalb eines Zeitintervalls t nicht mehr Kosten erzeugt werden können, als die Größe des Intervalls selbst. Ausgehend vom aktuellen Zeitpunkt **now** wird mit den verbleibenden Kosten des PGPs ein Checkpunkt $t = now + remaining$ eingerichtet, bei dem die Kosten der Threads mit **checkCost()** überprüft werden. Wenn die Kapazität bereits erschöpft ist, wird der **overrunHandler** des PGP-Objekts aufgerufen. Wenn sie jedoch noch nicht vollständig verbraucht ist, dann wird die restliche verbleibende Kapazität berechnet und der Überprüfungsvorgang wiederholt. Dabei wird ein neuer Checkpunkt t mit der aktuell verbleibenden Kapazität eingestellt.

Algorithm 3 COST-MONITORING FÜR PGPS INNERHALB DER JAMAICAVM

```
1: for each PGP-period do
2:    $remaining \leftarrow cost$ 
3: label:
4:   at  $t = now + remaining$  do  $remaining \leftarrow checkCost()$ 
5:   if  $remaining > 0$  then
6:      $remaining = remaining - spentCosts$ 
7:     goto label
8:   else
9:     call overrunHandler
10:  end if
11: end for
```

Der in Algorithmus 3 beschriebene Vorgang hat die Eigenschaft, dass die Kosten-Checks frequenter werden, wenn die verbrauchte Kapazität sich ihrer Grenze nähert. Die Problematik hierbei ist, dass dieser Mechanismus bei jeder PGP-Periode neu startet und dass es keine Möglichkeit gibt mit der PGP-Kapazität dynamisch aufgefüllt werden kann.

Die Perioden-Anfänge der PGP-Objekte werden in der JamaicaVM durch Timer realisiert. Für die Implementierung der CBS in dieser Arbeit wurden zunächst diese Timer deaktiviert, um die periodische Aufladung der PGP-Kapazität umzugehen. Da bei den Constant Bandwidth

Servern keine Vorgänge zu Beginn ihrer Perioden notwendig sind, sind diese nur logisch und dienen der Bestimmung ihrer Bandbreite und ihrem Scheduling. Des Weiteren wurden die RTSJ-Schnittstellen der `ProcessingGroupParameters` mit zwei neuen Funktionen erweitert:

```
void setRemainingCost(RelativeTime cost);  
  
RelativeTime getRemainingCost();
```

Mit diesen Anpassungen ist die Kapazitäts-Steuerung der PGPs nun völlig benutzerdefiniert. Bei `setRemainingCost(cost)` wird der übergebene Wert `cost` dem internen Parameter `remainingCost` zugewiesen und der Parameter `currentCost = 0` gesetzt. Diese Methode ermöglicht eine instantane Aufladung der PGP-Kapazität, so wie sie von einem Constant Bandwidth Server bei einer vollständigen Erschöpfung seiner Kapazität (CBS-Regel 4) oder bei seiner Aktivierung (CBS-Regel 7) benötigt wird (vgl. CBS-Regeln in Kapitel 3.2.2, auf Seite 89). Eine sofortige Kapazitäts-Entleerung kann ebenfalls leicht realisiert werden, diese wird z.B. bei der Rekonfiguration bzw. Suspension eines CBS an einem Modus-Wechsel vollzogen (vgl. Kapitel 4.4.6). Zuletzt wird die Kostenüberwachung mit der neuen Kapazität ab dem Zeitpunkt (`now`) des Aufrufs von `setRemainingCost(cost)` gestartet und auf die gleiche Weise wie in Algorithmus 3 durchgeführt.

In `getRemainingCost()` wird ein Kosten-Check innerhalb der JVM betrieben, um die verbleibende Kapazität eines PGPs zu ermitteln. Dabei wird nicht nur der Wert von `remainingCost` zurückgegeben, da der Funktionsaufruf zwischen zwei Check-Punkten auftreten kann. Damit wäre der zurückgegebene Wert nicht aktuell. Stattdessen werden in `getRemainingCost()` die tatsächlichen Prozessor-Zyklen der Threads seit dem Zeitpunkt der letzten Kapazitätsaufladung ausgelesen und zusammengefasst in Millisekunden zurückgegeben.

Mit diesen simplen Erweiterungen der RTSJ 1.0.2 Klassen-Bibliotheken war es ohne Modifikation der JamaicaVM möglich, die PGPs als Basis für die Implementierung von CBS zu etablieren. Diese Schnittstellen können jedoch generisch für die Implementierung beliebiger Server-Mechanismen verwendet werden. Sie wurden bereits mit der RTSJ Expert Group diskutiert und in einer ähnlichen Form in die Definitionen der RTSJ 2.0 `ProcessingGroups`¹⁹ übernommen.

4.5.3 Benutzerdefiniertes Scheduling von Echtzeit-Threads

Der in Kapitel 4.4.3 verwendete Prioritäts-Skalierungsmechanismus für die Implementierung eines benutzerdefinierten (EDF) Scheduling auf der Task-Ebene wurde bereits in einer ähnlichen

¹⁹RTSJ 2.0 ersetzt die alten PGPs durch neue, semantisch klarer definierte `ProcessingGroups`

Form von Zerzelidis et al. in [130, 131] vorgeschlagen. Die Motivation dieser Arbeit war, dass obwohl RTSJ intendiert wurde, mehrere austauschbare Scheduler zu unterstützen, die aktuelle Spezifikation nur den `PriorityScheduler` beinhaltet und die RTSJ API keine standardisierten Mechanismen für die Implementierung von anderen (applikationsspezifischen) Scheduling-Verfahren bereitstellt. Mit einer Erweiterung der API sollen nun unterschiedliche Mechanismen auf der Applikations-Ebene gleichzeitig unterstützt werden.

Der hauptsächliche Unterschied zu dem in dieser Promotionsarbeit entwickelten Mechanismus auf der Task-Ebene besteht darin, dass das in [130, 131] entwickelte Modell Scheduling-Vorgänge bei allen potentiell blockierenden Systemaufrufen²⁰ durchführt. Wenn also ein aktuell aktiver Thread beispielsweise bei einem Systemaufruf oder einer I/O-Operation wartend blockiert, dann soll die Task-Menge durch den Scheduler neu-geplant und der am nächsten geeignetste Thread ausgeführt werden. Dafür benötigt das Modell jedoch eine Unterstützung durch die unterliegende RT-JVM und erfordert Anpassungen innerhalb der RTSJ Klassen-Bibliotheken. Dabei müssen auf allen Ebenen einer RTSJ-Implementierung (JVM und Klassenbibliothek) Funktionen, die potentiell blockierende OS-Aufrufe tätigen, durch Call-Back Funktionen (`prepareToSuspend()` und `reschedule()`) für den Scheduler umhüllt werden. Dieser wird somit vom ausführenden Thread jeweils vor und nach einer blockierenden Operation mit einer Scheduling-Priorität aufgerufen und bestimmt den nächsten zu aktivierenden Thread. Die Priorität dieses Threads wird dann, ähnlich wie beim Mechanismus in dieser Arbeit, entsprechend angehoben, während alle anderen Aktivitäten mit niedrigeren Prioritäten verdrängt werden.

Im Gegensatz dazu wurden die Scheduling-Punkte auf der Task-Ebene in dieser Arbeit allein um die periodischen Suspendierungen der Echtzeit-Threads in `waitForNextPeriod()` realisiert — d.h. bei jeder Thread-Aktivierung und am Ende einer jeden Thread-Periode. Somit waren für das Scheduling auf der Task-Ebene keine Erweiterungen der RTSJ-Bibliotheken und Anpassungen innerhalb der verwendeten RT-JVM notwendig. Wie in Kapitel 4.4.3 gezeigt wurde, ist die Überprüfung der Aktivierungsreihenfolge der Threads an ihren (periodischen) Suspendierungs- und Aktivierungs-Zeitpunkten genügend, um anhand ihrer absoluten Zeitschranken einen gültigen EDF-Plan zur Laufzeit zu generieren (siehe Beispiel in Abbildung 4.4).

Die Berücksichtigung der Blockierungs-Effekte der Threads innerhalb des Prioritäts-Skalierungsmodells ist nicht ausschlaggebend für die Einhaltung ihrer EDF-Reihenfolge. Wenn Threads blockieren, dann sorgt der etablierte Mechanismus anhand des darunterliegenden prioritäts-basierten FIFO-Schedulings innerhalb des Betriebssystems automatisch für eine ordnungsgemäße Ausführung. Dies wird dadurch gewährleistet, dass zu jedem Zeitpunkt innerhalb

²⁰wie z.B. `sleep`, `read`

eines Prioritäts-Bands nur ein Thread mit einer **ACTIVE**-Priorität existiert (siehe Kapitel 4.4.4). Da die Prioritäts-Bänder über den RTSJ **PriorityScheduler** direkt auf den POSIX FIFO-Scheduler des darunterliegenden (Linux-)Betriebssystems abgebildet werden (vgl. Kapiteln 4.3.2 und 4.3.3), greift dort bei einer Blockierung die durch **SCHED_FIFO** vorgegebene Strategie (siehe POSIX Part 1: System Application Program Interface [121], Kapitel 13.2.1). Diese etabliert sich im Prioritäts-Skalierungsmodell wie folgt:

- Wenn ein Echtzeit-Thread mit einer **ACTIVE**-Priorität blockiert, dann wird ihm die CPU entzogen — er wird temporär von seiner Prioritäts-Warteschlange entfernt als „blockiert“ zwischengespeichert.
- Da kein weiterer aktiver Thread im Prioritäts-Band existiert, können Threads mit niedrigeren **PREEMPTED**-Prioritäten ausgeführt werden. (Das selbe gilt auch für Threads niedrigerer Prioritäts-Bänder.)
- Diese werden in ihren Warteschlangen mit einer FIFO-Reihenfolge optimistisch abgearbeitet und unterliegen keinen zeitlichen Garantien. Letztere werden nur durch die Betrachtung der Zeitschranken und die Priorisierung an den Scheduling-Punkten gewährleistet.
- Wenn der zuvor blockierte Thread wieder lauffähig wird, wird er am Ende seiner **ACTIVE**-Warteschlange eingefügt [121]. Da er der einzige Thread innerhalb dieser Priorität ist, bekommt er sofort wieder die CPU und kann weiter ausführen.
- Dabei wird ein unter Umständen gerade ausführender niedrig-priorer **PREEMPTED**-Thread sofort verdrängt.

Mit diesem Mechanismus kann ein aktiver Thread bei einer Blockierung nicht durch einen anderen aktiven Thread mit einer späteren Frist behindert werden, so dass er seine eigene Frist verpasst. Wird die Priorität eines temporär blockierten Threads verändert (z.B. bei einer Verdrängung), so kann dies nur aufgrund der Aktivierung eines anderen Threads, mit einer kleineren Frist, passieren. Der ursprünglich aktive Thread deblockiert dann zu einem späteren Zeitpunkt mit einer **PREEMPTED** Priorität, so als wäre er „ganz normal“ verdrängt worden. Somit sorgt der Mechanismus automatisch für die Einhaltung der absoluten Deadlines der Tasks, auch wenn diese temporär blockieren. Eine explizite Behandlung der Blockierungs-Effekte der Threads hat keine Implikationen auf das Erreichen ihrer Deadlines, erzeugt jedoch einen höheren Scheduling-Overhead im System.

5

Evaluation

In diesem Kapitel werden die erarbeiteten Scheduling-Mechanismen anhand ihrer RTSJ-basierten Implementierung im ARTOS-Framework evaluiert. Dabei soll die Wirksamkeit der entwickelten Methoden für den Umgang mit dynamisch gestarteten und gestoppten Echtzeit-Anwendungen und ihren unbekannten und zur Laufzeit schwankenden Ressourcenanforderungen nachgewiesen werden. Hierfür werden in erster Linie die Funktionen des hierarchischen Schedulers innerhalb verschiedener Test-Szenarien näher untersucht und ihre Auswirkungen auf das Verhalten eines realen System-Prototyps mit den ursprünglichen Anforderungen aus Kapitel 1.3 verglichen. Die Mechanismen von ARTOS müssen dafür sorgen, dass die verfügbaren Ressourcen innerhalb der Plattform stets optimal ausgeschöpft werden, dabei der höchstmögliche Nutzen aus den laufenden Anwendungen gewonnen wird und temporale Fehler in Form von Fristverletzungen soweit wie möglich minimiert oder ganz vermieden werden. Die Evaluation findet innerhalb einer konkreten Ausführungsplattform statt, deren genaue Konfiguration zunächst vorgestellt wird. Für die verschiedenen Testszenarien wurden künstliche Echtzeit-Anwendungen entwickelt, die ein vorgegebenes, konfigurierbares Ausführungsverhalten umsetzen können. Ihr Ressourcenverbrauch sowie bestimmte Vorgänge innerhalb des Frameworks werden dann numerisch erfasst und mit speziellen Werkzeugen näher untersucht. Auf diese Weise können die Funktionen des

Schedulers auf allen Ebenen evaluiert werden. Der Fokus liegt dabei zunächst auf der Approximation des Verhaltens von Anwendungen und auf einem korrekten, zur Laufzeit erzeugten Ablaufplan für ihre Aktivitäten. Danach werden die automatischen Qualitätseinstellungen der Anwendungen und ihrer Tasks im Rahmen künstlich erzeugter Engpass-Szenarien getestet und das Verhalten des Systems bei zyklisch schwankenden Ressourcenverbräuchen untersucht. Bei einer expliziten Evaluation der Optimierung werden daraufhin die Performanz-Grenzen eines auf den aktuellen Prototyp aufbauenden ARTOS-Systems in mehreren Dimensionen numerisch erfasst. Zum Schluss wird die Implementierung einer ARTOS-kompatiblen Video-on-Demand (VoD) - Anwendung vorgestellt, mit welcher der durch das Scheduling-Modell erzeugte Qualitätsgewinn sowohl quantitativ als auch subjektiv evaluiert werden kann. Ferner dient die VoD-Anwendung als Demonstrator für den entwickelten Prototyp und die Konzepte von ARTOS. Bei allen Evaluations-Szenarien sind Faktoren wie Auftritte von Fristverletzungen oder der Verlauf der Task-Kosten und ihrer Schätzungen maßgeblich für den Nachweis einer korrekten Funktion des Systems.

5.1 Evaluations-Umgebung

Im Folgenden wird die Konfiguration der verwendeten Evaluations-Plattform, die Realisierung der künstlichen Anwendungen mit ihren einstellbaren Lastprofilen und die zur Auswertung der Ergebnisse verwendeten Werkzeuge und Mechanismen vorgestellt.

5.1.1 Plattform-Konfiguration

Die im Verlauf des Kapitels vorgestellten Messungen und Test-Szenarien wurden auf einem 64-Bit Fujitsu ESPRIMO P920 Desktop-System mit einem Intel(R) Core(TM) i7-4770 Prozessor, 16 GB RAM und 500 GB HDD betrieben. Der i7-4770 Prozessor besitzt vier mit bis zu 3.40 GHz getakteten Rechenkernen mit jeweils zwei Hyper-Threads, 256 KB Level 2 Instruktions- und Daten-Cache und 15 MB geteilten Last Level Cache (LLC). Als Betriebssystem wurde Ubuntu Linux 14.04.2 LTS mit der Kernel-Version 3.14.23-rt20 verwendet. Der Kernel wurde mit dem entsprechenden CONFIG_PREEMPT_RT Patch in der Version 3.14-rt erweitert, um mit minimalen Latenzzeiten bei Systemaufrufen und entsprechenden mit Prioritätsvererbung versehenen Lock-Primitiven Echtzeitanwendungen unterstützen zu können (vgl. Kapitel 4.3.4). Die automatische Skalierung der Taktrate der CPU Kerne durch das Betriebssystem wurde deaktiviert, um durch die Hardware-Plattform bedingte Performanz-Schwankungen der Anwendungen auszuschliessen.

Auf dem echtzeitfähigen Linux-Kernel setzt eine mit den im Kapitel 4.5.2 vorgestellten Erweiterungen für Prozess-Gruppen angepasste Version der JamaicaVM 6.4 Release 4 (64bit) auf. Diese implementiert die RTSJ Version 1.0.2. Bei den Tests wurde die Singlecore-Variante der JVM verwendet, die Cost-Monitoring [123] für einzelne Echtzeit-Threads oder ganzer Thread-Gruppen unterstützt. Die Cost-Monitoring Genauigkeit der JVM wurde dabei auf $10\ \mu\text{s}$ eingestellt. Da die Kosten der bei den Test-Szenarien involvierten Anwendungs-Tasks sich im Millisekunden-Bereich erstrecken, ist diese Präzision für die Evaluation ausreichend. Das Cost-Enforcement der JVM wurde nicht verwendet, da die entwickelten CBS-Mechanismen und ihr Scheduling innerhalb des Frameworks selbst eine Durchsetzung von Kapazitäten für die gekapselten Anwendungen bereitstellen. Die JVM wurde auf einem von anderen Anwendungen und System-Diensten abgeschotteten Prozessor-Kern ausgeführt, um Interferenzen mit anderen Prozessen zu vermeiden. Der RTSJ Prioritätsraum wurde auf einem freien Bereich der FIFO-Prioritäten 49..77 im Kernel abgebildet. Darüber hinaus wird beim Start der JVM genügend Heap-Speicher alloziert und die initiale Heap-Größe so eingestellt, dass möglichst wenig Garbage-Collections stattfinden. Die JamaicaVM erlaubt bei Bedarf, den Garbage-Collector (GC) in einem separaten Thread auszulagern, welcher mit einer festen RTSJ-Priorität ausgeführt wird. Bei den Experimenten wurde der GC mit einer nicht-echtzeit Priorität gleich Null konfiguriert, um die Echtzeit-Tasks der Anwendungen und innerhalb des Frameworks nicht zu behindern. Dementsprechend wird der Speicher bei den Experimenten nur dann aufgeräumt, wenn Ressourcen dafür übrig bleiben. Die JamaicaVM wurde ausschließlich im Interpretier-Modus verwendet. Eine Ahead-Of-Time Compilation des Frameworks und der Anwendungen wurde nicht angewandt, weil die JVM innerhalb nativ kompiliertem Code kein Class-Loading unterstützt. Dieses ist jedoch für das dynamische Installieren und Starten von in Bundles gekapselten Anwendungen zwingend nötig.

Die nativen Teile der Anwendungen und des ARTOS-Frameworks wurden mit der GNU Compiler Collection (GCC) 4.8.2 übersetzt und gegen die GNU-C-Bibliothek (Glibc) 2.19.90 gelinkt. Bei der Konfiguration handelt es sich somit um ein Standard Linux-System, welches gleichermaßen innerhalb eines Servers, eines Desktop-Systems oder eines eingebetteten bzw. ressourcen-eingeschränkten Geräts bereitgestellt sein kann.

5.1.2 Evaluations-Werkzeuge

Online-Visualisierung des Systemzustands

Zur Online-Überwachung des Systemzustands und des Anwendungsverhaltens innerhalb einer ARTOS-Plattform wurde ein spezielles Visualisierungs-Tool entwickelt, welches in Abbildung 5.1

dargestellt ist. Dieses empfängt Informationen über ein Socket und kann somit auf einem entfernten Rechner betrieben werden, um das beobachtete System nicht zu beeinflussen. Ein dazugehöriges Backend im ARTOS-Framework bereitet kontinuierlich Daten über den aktuellen Systemzustand auf und sendet diese zur Visualisierungs-Anwendung.

Abbildung 5.1: Visualisierung des Zustands eines ARTOS-Systems

Das Visualisierungs-Tool zeigt zunächst Informationen über die einzelnen Applikationen an, wie z.B. ihre aktuelle Priorität, ihre Modi, die totale Anzahl der aufgetretenen Fristverletzungen und der durch Fehlschätzungen bedingten Kosten-Überläufen. Für jeden Anwendungsmodus wird auch die bereits ermittelte notwendige Ressourcenbandbreite angegeben. Nur für den aktuell aktiven Modus einer Anwendung werden zusätzliche Informationen über den Zustand ihrer Tasks und ihres CBS angezeigt. Dazu gehören Angaben über die aktuellen Task-Modi, ihre Periodenlängen, die Grenzen ihrer Sicherheitspuffer, ihre mittleren Kosten und Standardabweichung, die aktuelle absolute Task-Deadline sowie Angaben über ihre aktuelle logische Priorität und Ausführungsstatus. Die CBS-Informationen zeigen die aktuelle modus-abhängige Periode eines Servers, seine maximale und die aktuell verbleibende Kapazität, die belegte Ressourcenbandbreite, die aktuelle Server-Frist, die Anzahl an Deadline-Verschiebungen und Server-Überläufen sowie die Planbarkeit der anwendungs-lokalen Task-Menge und die aktuelle Server-Priorität. Darüber hinaus werden im unteren Abschnitt des Visualisierungs-Fensters globale System-Informationen dargestellt, wie die aktuelle logische Systemzeit, der Zeitpunkt und der Grund der letzten Rekonfiguration des Systems, die aktuellen Kosten der durch das Framework betriebenen Scheduling-Vorgänge sowie die totale Bandbreiten-Auslastung im System. Die globale logische Zeit wird beim Start des Frameworks mit der durch RTSJ bereitgestellten standardmäßigen `RealtimeClock` synchronisiert und dient als Basis für das interne Scheduling im

Framework, insbesondere für den Vergleich der Task- bzw. CBS-Deadlines. Sie wird in Millisekunden angegeben.

Das Beispiel in Abbildung 5.1 zeigt grob die Wirkungsweise des Scheduling in einem realen ARTOS-System. Zwei gleich priorisierte Anwendungen mit jeweils zwei Modi werden in ihrem jeweiligen Modus 1 ausgeführt; sie lasten dabei das System mit 98.2 % möglichst optimal aus. Bei keiner der Anwendungen sind soweit Fristverletzungen aufgetreten. Bei diesem Beispiel wurde der durch den jeweiligen Anwendungsmodus erzeugte Nutzen $U_i(M_{i,j}) = j$ gleich seinem Index innerhalb seiner Anwendung gesetzt. Der aktuelle Gesamtnutzen im System ist also bei dieser Momentaufnahme gleich 2. Die Optimierung von ARTOS hat eine gleichmäßige Verteilung der Ressourcen und somit Aktivierung mehrerer Anwendungen vorgezogen, statt den gleichen Nutzen durch eine Anwendung in einem höheren Modus zu erlangen (vgl. Kapitel 3.2.3). Eine höhere Priorisierung einer Anwendung hätte demnach für sie eine sofortige Modus-Erhöhung zufolge, weil der generierte Nutzen dann in jedem Fall größer werden würde. Die andere Anwendung würde dabei in ihren Null-Modus deaktiviert werden, da sie mit keinem anderen Modus innerhalb der übrigen Ressourcen eingebracht werden könnte. Dieses Beispiel verdeutlicht die Fähigkeit von ARTOS, verfügbare Ressourcen möglichst optimal auszulasten, dabei eine gleichmäßige und prioritätsgesteuerte Verteilung unter den Anwendungen zu etablieren, wobei stets der höchstmögliche Nutzen angestrebt wird.

Kommando-Shell

Das ARTOS-Framework kann mit Hilfe eines Skripts aus einer Linux-Shell gestartet werden. Darin werden zunächst Einstellungen der JamaicaVM vorgenommen, wie z.B. der zu reservierende Heap-Speicher, die Abbildungen der RTSJ Prioritäten auf Betriebssystem-Prioritäten sowie die Pfade für die zu ladenden nativen Bibliotheken. Zum Schluss wird die Hauptklasse des Frameworks aufgerufen. Die JVM muss mit Superuser-Rechten ausgeführt werden, damit sie Zugriff auf den gesamten FIFO-Prioritätsraum im Kernel bekommt. Der Boot-Vorgang des Frameworks ist in Abbildung 5.2a dargestellt. Das Framework liest automatisch eine Konfigurationsdatei `init.xargs` aus, in welcher Anweisungen für die Initialisierung des Systems sowie die zu installierenden und zu startenden Bundles angegeben werden.

Für die Kontrolle des Frameworks zur Laufzeit wurde eine einfache Shell-Anwendung entwickelt, welche in einem OSGi-Bundle verpackt und beim Boot-Vorgang des Frameworks automatisch gestartet wird. Die ARTOS-Shell enthält zunächst zwei Kommando-Gruppen: 1) für die Steuerung des zugrunde liegenden OSGi Systems und 2) für die Überwachung und Kontrolle der ARTOS-kompatiblen Echtzeit-Anwendungen (siehe Abbildung 5.2b). Mit den Befehlen aus der

Abbildung 5.2: Steuerung eines ARTOS-System mit einer Kommando-Shell

ersten Gruppe können zum Beispiel OSGi-Komponenten installiert, gestartet, aktualisiert und gestoppt werden, sowie ihre Abhängigkeiten verwaltet werden. Die zweite Gruppe stellt Kommandos für die Priorisierung der Anwendungen, ihre explizite Aktivierung in unterschiedlichen Modi und für die Überwachung ihrer Ausführungs-Stati bereit. Ebenso können Anwendungen in bestimmten Modi befestigt („gepinnt“) werden, womit die Ausführung eines bestimmten Modus durch den Benutzer oder das System erzwungen werden kann. Dabei werden die übrigen Modi der betroffenen Anwendung auch vor der Optimierung „versteckt“, damit sie nicht automatisch in diese umgeschaltet werden kann. Eine in einem bestimmten Modus gepinnte Anwendung wird im Visualisierungs-Tool entsprechend markiert, Rekonfigurationen dieser Anwendung werden durch das Framework unterbunden. Auf diese Weise können die automatischen Qualitätsanpassungs-Mechanismen von ARTOS für bestimmte Anwendungen ausgeschaltet werden und somit gezielt Engpass-Situationen hergestellt werden, was später bei manchen Evaluations-Szenarien ausgenutzt wird.

Offline-Analyse der Task-Kosten

Zur Analyse des Anwendungsverhaltens können bei Bedarf die im Framework intern gemessenen Kosten und die etablierten Schätzungen ihrer Tasks in Log-Dateien numerisch ausgegeben und

später offline visualisiert werden. Die Log-Daten werden im Framework gepuffert und nur bei Verfügbarkeit von freien Ressourcen in entsprechenden Dateien geschrieben. Zur Visualisierung der Werte wurde Gnuplot in der Version 4.6 verwendet.

Abbildung 5.3: Plot der Kosten einer Task

Abbildung 5.3 zeigt ein Beispiel der Visualisierung des gemessenen Verhaltens einer Task mit einem konstant schwankenden Ressourcenverbrauch. Neben den gemessenen Task-Kosten pro Periode (**Measured**) werden auch ihre exponentiell geglätteten Werte (**Mean**), die aktuelle Standardabweichung (**SD- σ**) und die aktuelle Schätzung (**Estimate**) der Kosten dargestellt. Wie bereits in Kapitel 3.2.1 erklärt, ergibt sich eine neue Schätzung aus der doppelten Standardabweichung über den aktuellen Mittelwert der Kosten, und zwar dann, wenn die letzte Schätzung und somit der Sicherheitspuffer der Task durch die mittleren Kosten verletzt wird. Wie man an dem Beispiel in Abbildung 5.3 sehen kann, wird die Schätzung der Task nach ihrem Start vier Mal aktualisiert, bis sie sich bei einem entsprechenden Wert von ca. 100 ms stabilisiert.

Jamaica-Trace

Die JamaicaVM stellt ein Tracing-Tool zur Verfügung, das auf ähnliche Weise wie das für den Zustand eines ARTOS-Systems entwickelte Visualisierungs-Tool funktioniert und JVM-interne Ereignisse über ein Socket zu einer Darstellungs-Anwendung übermittelt. Mit dem Jamaica-Trace Tool kann unter anderem die Aktivierung von Echtzeit-Threads und asynchronen Event-Handlern innerhalb der JVM zeitlich nachverfolgt und analysiert werden.

Abbildung 5.4: Jamaica-Trace

Über eine Eingabe-Maske (siehe Abbildung 5.4a) können vor einem Testlauf die gewünschten zu beobachtenden Ereignisse und Informationen innerhalb der JVM ausgewählt werden. Dazu gehören insbesondere die Ausführungszustände der Threads, ihre Namen und RTSJ-Prioritäten. Darüber hinaus können aus einer ausgeführten Anwendung über spezielle JVM-Schnittstellen benutzerspezifische Ereignisse zum Tracing-Tool versendet werden. Die JamaicaVM muss beim Start für die Ausgabe dieser Informationen explizit mit einer Verbindung zur Trace-Anwendung konfiguriert werden. Wenn die Aufnahme der Ereignisse gestartet wird, sendet die JVM kontinuierlich die gewünschten Informationen zur Trace-Anwendung, wo diese zunächst zwischengespeichert werden. Wenn die Aufnahme beendet wird, wird ein Auswertungs-Fenster angezeigt (siehe Abbildung 5.4b), mit dem die Aktivierung und der zeitliche Ablauf aller Aktivitätsträger innerhalb der JVM nachverfolgt werden kann¹. Dieses Werkzeug wird später zur Analyse des applikations-lokalen und -übergreifenden Scheduling verwendet, um die Korrektheit des durch ARTOS generierten Ablaufplans auf der Task-Ebene nachzuweisen.

¹Dabei wird z.B. die Zeit in der ein Thread aktiv ist mit einer roten Linie markiert (siehe Abb. 5.4b).

5.1.3 Künstliche Anwendungen

Bei den Experimenten werden neue Applikationen innerhalb OSGi-Bundles gekapselt und innerhalb der Plattform bereitgestellt und gestartet. Zur Evaluation des Scheduling-Modells mit unterschiedlichen Last-Szenarien, wurde eine generische Test-Anwendung entwickelt, mit der künstliche ARTOS-Applikationen mit mehreren Tasks und unterschiedlichen Betriebsmodi konfiguriert werden können. Sie werden im ARTOS-System zunächst mit einer niedrigen Priorität gestartet und initial ausgemessen. Dadurch entstehen erste Bandbreiten-Werte für alle Anwendungsmodi, welche bei späterer Aktivierung verfeinert werden.

Die künstlichen Applikationen erzeugen gezielt Last im System, d.h. ihre Tasks verbrauchen eine konfigurierbare Menge an Ressourcen pro Periode. Auf diese Weise lassen sich Szenarien abspielen, bei denen Aktivitäten ein bestimmtes Burst- und Jitter-Verhalten aufweisen. Damit kann analysiert werden, wie oft und aufgrund welcher Faktoren Rekonfigurationen oder Fristverletzungen im System auftreten. Das Verhalten der Tasks kann also für jeden Task-Modus über die gängigen Perioden T , Fristen D und Kosten C hinaus mit den folgenden zusätzlichen Parametern eingestellt werden:

- C_{nom} , C_{max} : nominale und maximale Kosten der Task (CPU-Zeit in ms)
- B_{prd} , B_{dur} : die Periode und die Dauer von Bursts (in Task-Perioden)
- J_{wnd} : Jitter-Fenster der Kosten in Prozent um die vorgegebenen C_{nom} , C_{max}

Abb. 5.5 zeigt die Aktivierung einer Task τ mit den folgenden Vorgaben (in ms):

$$\tau = (T = 300, C_{nom} = 50, C_{max} = 90, D = 200, B_{prd} = 20, B_{dur} = 10, J_{wnd} = 20).$$

Die Auswirkungen der Parametrisierung der Task auf ihr Laufzeitverhalten werden in der Abbildung verdeutlicht. Dabei wurden, wie im letzten Kapitel 5.1.2 beschrieben, die gemessenen und die mittleren Kosten der Task, die aktuelle Standardabweichung, die sich daraus ergebenden laufenden Kostenschätzungen (**Cost**) = $c'_k + 2 \cdot \sigma$ und die dynamische Obergrenze des Sicherheitspuffers (**Estimate**) aus dem Framework extrahiert und mit Gnuplot visualisiert. Der Einfachheit halber wurde die untere Schranke des Sicherheitspuffers $c'_k - 2 \cdot \sigma$ nicht in die Darstellung einbezogen. Die Task-Kosten steigen alle 20 Perioden auf den Wert von C_{max} ² für eine Dauer von 10 Perioden bis sie schließlich wieder auf C_{nom} abfallen. Das Jitter-Fenster von 20% bezieht sich auf den jeweiligen aktuellen Vorgabewert C_{nom} oder C_{max} .

²Hierbei wird ein rampenartiger Anstieg emuliert, welcher im Normalfall der anspruchsvollste ist.

Abbildung 5.5: Beispiel einer periodisch burstenden Anwendung in ARTOS (aus [15])

Abbildung 5.6: Generische Test-Anwendung

In Abbildung 5.6 ist die Eingabemaske der generischen Test-Anwendung dargestellt, in der Konfigurationen von ARTOS-Anwendungen mit mehreren Tasks und Betriebsmodi erstellt, bearbeitet, gestartet, als Datei gespeichert und wieder geladen werden können. Jede Anwendung besteht aus Definitionen von Betriebsmodi und Tasks. Jede Task kann mit mehreren Modus-Definitionen und unterschiedlichen Parametervorgaben versehen werden. Bei den Modi-Definitionen der Anwendungen werden die Task-Modi zu Anwendungsmodi kombiniert. Bei einem Start-Befehl werden alle Anwendungen und ihre Tasks nach einer kurzen Initialisierungs-Phase synchron gestartet. Dabei werden zunächst alle Definitionen in zugehörige **Task**-, **TaskMode**-, **ApplicationMode**- und **Application**-Instanzen überführt, miteinander verknüpft und entsprechend parametrisiert. Die Ausführung der Anwendungen kann dann über die Kommando-Shell des Frameworks gesteuert werden. Die Implementierung der Task-Modi innerhalb der Testanwendung ist generisch und wird in Listing 5.1 exemplarisch dargestellt. Jede Task-Modus Definition wird als eine **GenericTaskMode**-Instanz realisiert, die automatisch den Vorgaben aus der Eingabemaske der Test-Anwendung folgt.

In einer **for**-Schleife (Zeile 35) erzeugt die Task durch Inkrementieren eines Zählers reale Rechenlast, bis die vorgegebenen Kosten im entsprechenden Modus erreicht werden. Dabei wird bei jeder Schleifeniteration die bereits verbrauchte CPU-Zeit innerhalb der aktuellen Task-Periode gemessen und mit einem Zielwert verglichen. Auf diese Weise können die Scheduling-Methoden, unabhängig von der Rechen-Performanz der unterliegenden Plattform, getestet werden.

Der Zielwert richtet sich nach dem aktuellen Burst-Zustand der Task (Zeile 16) und somit nach dem entsprechenden Wert für C_{nom} bzw. C_{max} (Zeilen 21 und 23). Darüber hinaus wird im vorgegebenen Jitter-Fenster J_{wnd} ein zufälliger Jitter-Wert für jede Task-Periode berechnet (Zeile 26), welcher von den jeweiligen Zielkosten zufällig abgezogen bzw. dazu addiert wird (Zeilen 29 und 31). Da die Scheduling-Vorgänge auf der Task-Ebene, die durch die Tasks selbst verrichtet werden, mit zusätzlichen Kosten verbunden sind, werden diese im Framework intern gemessen und statistisch erfasst. Diese Umgebungskosten werden dann von den zu erreichenden Zielkosten einer jeden Task abgezogen (Zeile 33), um ihre Lastvorgaben möglichst präzise zu erfüllen. Letztendlich wird die Burst-Bedingung einer Task solange aufrecht erhalten, und somit ihre Zielkosten mit C_{max} initialisiert, bis die Burst-Dauer von B_{dur} -Task-Perioden erreicht wurde. Danach wird C_{nom} weiter verwendet, bis die Burst-Bedingung in Zeile 16 erneut greift.

Das Task-Beispiel weiter oben in diesem Kapitel (auf Seite 185) wurde ebenfalls mit der generischen Test-Anwendung implementiert und in einem Test-Szenario abgespielt. Die gesammelten Ausführungs-Daten über die Task wurden danach mit Gnuplot visualisiert. Die Darstellung in Abbildung 5.5 zeigt die Gültigkeit der Implementierung der generischen Task-Modi. Die Task

verhält sich innerhalb der Test-Anwendung genau nach ihren Vorgaben und erzeugt das gewünschte Lastprofil. Szenarien dieser Art werden nun nach genauen Vorgaben abgespielt, um gezielt die Funktionen des hierarchischen Schedulers auf allen seinen Ebenen zu evaluieren.

```
1 public class GenericTaskMode extends TaskMode {
2     int periodsCount = 0, int BdurCount = 0;
3     boolean burstCondition = false;
4     int Cnom, Cmax, Brate, Bdur, Jwnd;
5     int jitter = 0, jitterBound;
6     Random random = new Random(1234);
7     ...
8
9     public void run(AsynchronouslyInterruptedException exception)
10         throws AsynchronouslyInterruptedException {
11
12         long start = measurement.getCpuTimeMillis();
13
14         periodsCount++;
15
16         if(periodsCount % Brate == 0)
17             burstCondition = true;
18
19         if (burstCondition) {
20             BdurCount++;
21             attainedCosts = Cmax;
22         } else {
23             attainedCosts = Cnom;
24         }
25
26         jitter = random.nextInt(Jwnd);
27
28         if(random.nextBoolean())
29             attainedCosts = attainedCosts + jitter;
30         else
31             attainedCosts = attainedCosts - jitter;
32
33         long end = (long)Math.floor(attainedCosts - getTask().EnvCostsEstimator.costs);
34
35         for(counter = 0; measurement.getCpuTimeMillis() - start < end; counter += 1);
36
37         if (BdurCount == Bdur) {
38             burstCondition = false;
39             BdurCount = 0;
40         }
41     }
42 }
```

Listing 5.1: Implementierung eines generischen Task-Modus

5.2 Evaluation des Schedulings auf der Task-Ebene

In diesem Abschnitt wird das applikations-lokale EDF-Scheduling auf der Task-Ebene evaluiert. Nachdem RTSJ nur ein FIFO-Scheduling mit festen Prioritäten bereitstellt, wurde im Kapitel 4.4.3 ein Mechanismus entwickelt, mit dem EDF auf der Basis des RTSJ-Schedulers emuliert werden kann. Dafür wurde innerhalb des ARTOS-Modells für jede Applikation ein lokaler Task-Scheduler (vgl. Listing 4.8) bereitgestellt, welcher von den Applikations-Tasks zu bestimmten Zeitpunkten synchron oder asynchron aufgerufen wird und ihre Prioritäten zielgerichtet modifiziert, um einen EDF-konformen Ablaufplan zu erzeugen.

Bei diesem Evaluations-Szenario wird der in Kapitel 2.1.1 vorgestellte theoretische EDF-Ablauf zweier Tasks $\tau_i = (T_i, C_i, D_i)$ mit $\tau_1 = (5, 2, 4)$ und $\tau_2 = (3, 1, 2)$ mit ihrer tatsächlichen Aktivierung im ARTOS-System verglichen. Die Tasks wurden dabei mit der generischen Test-Applikation modelliert und innerhalb derselben „virtuellen“ ARTOS-Anwendung gekapselt. Auf diese Weise wird der Einfluss des CBS-Schedulings auf der Applikations-Ebene vermieden und nur das applikations-lokale Task-Scheduling getestet. Die Wirksamkeit der Test-Applikation wurde bereits im letzten Kapitel nachgewiesen, die Tasks erzeugen im System genau die durch die Anwendung vorgegebene Last. Wenn das Scheduling auf der Task-Ebene korrekt funktioniert, sollte also mit dem Tracing-Tool der JamaicaVM dieselbe Aktivierungsreihenfolge der Tasks, wie in ihrem theoretischen Ablaufplan, zu sehen sein. Zur besseren Übersichtlichkeit wird der theoretische Ablauf aus dem Beispiel in Abbildung 2.4 hier nochmal aufgeführt.

Abbildung 5.7: Theoretischer Ablaufplan des Task-Beispiels aus Kapitel 2.1.1

Da bei der aktuellen Java-basierten Implementierung allein die Scheduling-Umgebung auf der Task-Ebene einen durchschnittlichen Overhead von 2 - 3 ms erzeugt, wurden die Werte der Tasks aus dem Beispiel auf längere Zeitabstände skaliert. Die Definition der Task-Menge innerhalb der Eingabemaske der generischen Test-Anwendung ist in Abbildung 5.9 dargestellt. Wie dort

Handling und Task-Aktivierungen kommt (z.B. bei $t = 8s, 140ms$), wird die EDF-Reihenfolge der Tasks in jedem Fall eingehalten. Bei $t = 9s, 340ms$ beginnt die neue Hyperperiode der Tasks, die mit der vorherigen identisch ist und aus Übersichtsgründen hier nicht weiter gezeigt wird.

Abbildung 5.9: Konfiguration des Evaluations-Szenarios auf der Task-Ebene

Abbildung 5.10 zeigt eine Momentaufnahme des internen Systemzustands von ARTOS bei ca. 38 Sekunden nach Start des Experiments. Dies gleicht in etwa der Ausführung von 25 Task-Hyperperioden. Innerhalb dieser Zeit ist keine einzige Fristverletzung aufgetreten und die belegte Ressourcenbandbreite beträgt ca. 80,3 %. Dies entspricht in etwa dem auf Seite 93 theoretisch berechneten Bandbreiten-Wert $B_1 = 0.8$ für eine Anwendung mit der gleichen Task-Konfiguration. Der minimale Unterschied ist unter anderem durch Rundungsfehler und auch durch die leichte Überprovisionierung der Tasks im realen System zu erklären, welche durch minimale Schwankungen in ihrem Ressourcenverbrauch entstehen können (vgl. die Standardabweichung der Tasks in Abbildung 5.10).

Abbildung 5.10: Systemzustand beim Experiment

Nichtsdestotrotz, mit diesem Evaluations-Szenario wurde die Gültigkeit der entwickelten Mechanismen für das applikations-lokale EDF-Scheduling auf der Task-Ebene und die Korrektheit ihrer Implementierung nachgewiesen. Weitere Beispiele wurden aus Übersichtsgründen hier nicht aufgeführt. Nachfolgend soll das CBS-basierte Scheduling auf der Applikations-Ebene des hierarchischen Modells von ARTOS untersucht werden.

5.3 Evaluation des Scheduling auf der Applikations-Ebene

In diesem Abschnitt wird das CBS-Scheduling auf der Applikations-Ebene im Zusammenspiel mit dem applikations-lokalen Task-Scheduling evaluiert. Zur Erinnerung, das CBS-Scheduling benutzt dynamische Anwendungskapazitäten und verschiebt ganze Applikationen entsprechend zwischen zwei Prioritäts-Bändern, während die Task-Scheduler ihre Aktivitäten lokal steuern.

5.3.1 Eine Task pro Anwendung

In einem ersten Szenario werden die Tasks aus dem Evaluations-Beispiel im vorherigen Kapitel 5.2 innerhalb einzelner Applikationen A_1 und A_2 gekapselt und synchron ausgeführt. Die entsprechende Konfiguration der generischen Evaluations-Anwendung wird in Abbildung 5.11 dargestellt. Wie zu sehen ist, wurden hier die Parameter der Tasks erneut in Hunderten von Millisekunden hochskaliert und ihre Burst- und Jitter-Emulation deaktiviert. Zu erwarten ist, dass, wenn das EDF-basierte CBS-Scheduling auf der Applikations-Ebene korrekt funktioniert, ein ähnlicher Aktivierungs-Ablauf der Tasks wie in Abbildung 5.7 erzeugt wird.

Abbildung 5.11: Konfiguration des Evaluations-Szenarios auf der Applikations-Ebene

Betrachten wir zunächst die Konfigurationen der Anwendungs-Server, welche sich im Framework nach den vorgestellten Methoden aus Kapitel 3.2.2 ergeben. Dabei werden die CBS zunächst mit den kleinsten Task-Perioden innerhalb der Applikationen initialisiert, also $P_1 = 500$ ms und $P_2 = 300$ ms.

Abbildung 5.12: Kapazitätsberechnung der Applikations-Server

Für die entsprechende Kapazitätsberechnung der Server wird für jede Anwendung eine lokale Planbarkeitsanalyse betrieben. Das Ergebnis ist in Abbildung 5.12 dargestellt. Da für jede Anwendung die Hyperperiode gleich der jeweiligen Task-Periode ist, muss die Rechenzeitanforderung (DBF) jeweils nur für einen einzigen Deadline-Punkt berechnet und mit der Winkelhalbierenden (Time) verglichen werden. Der Schnittpunkt der Approximationsgerade $c_{D_1}(t)$ der ersten Anwendung mit der Winkelhalbierenden liegt bei $t'_1 = 66.6$ ms, der für die Approximationsgerade $c_{D_2}(t)$ der zweiten Anwendung bei $t'_2 = 50.0$ ms. Wie zu sehen ist, sind beide Anwendungen planbar. Wie in Kapitel 3.2.2 bereits erklärt wurde, ergibt sich die Kapazität eines Anwendungs-Servers aus dem Schnittpunkt der Approximationsgerade $c_{D_i}(t)$ mit der Hyperperiode der jeweiligen Task-Menge. Die entsprechenden Werte sind $R_1 = 240$ ms für Anwendung 1 und $R_2 = 133$ ms für Anwendung 2. Die Werte müssen hier nicht zusätzlich für die jeweiligen Server-Perioden der Anwendungen normiert werden, da letztere bereits den Hyperperioden entsprechen. Somit belegt A_1 theoretisch eine Bandbreite von $B_1 = 48.0$ % und A_2 eine Bandbreite von $B_2 = 44.4$ %. Das System ist also theoretisch zu 92.44 % ausgelastet.

Abbildung 5.13 zeigt eine Momentaufnahme des Systemzustands während der Ausführung beider Applikationen. Wie man sieht, weichen die allozierten Server-Kapazitäten aufgrund von geringfügigen Schwankungen der Task-Verbräuche leicht von den theoretisch berechneten Werten ab. Darüber hinaus rechnet das Framework intern mit `double` Werten, die bei der Visualisierung des Systemzustands als ganzzahlige Werte gerundet werden, was ebenfalls zu Abweichungen in der Darstellung führen kann. Die totale Auslastung des Systems ist somit zur Laufzeit mit 93.40 % ca. 1 % höher als der theoretisch berechnete Wert.

5 Evaluation

Abbildung 5.13: Systemzustand beim Experiment auf der Applikations-Ebene

Abbildung 5.14: Aktivierungsreihenfolge der Tasks

Abschließend betrachten wir nun in Abbildung 5.14 die tatsächliche Aktivierungsreihenfolge der Tasks im Jamaica-Trace Tool. Wie man sieht, ist diese innerhalb ihrer Hyperperiode mit ihrer theoretischen EDF-basierten Aktivierung in Abbildung 5.7 identisch. Demzufolge greift auch die EDF-basierte Aktivierung der Server. Dies wird vor allem auch durch die temporäre Verdrängung von A_1 durch A_2 zum Zeitpunkt $t = 5s, 120ms$ deutlich.

Wie man in der Darstellung der Anwendungszustände in Abbildung 5.13 erkennen kann, kamen bei diesem Szenario keine Kapazitäts-Überläufe innerhalb ihrer CBS vor, da die Tasks ihre Kosten-Vorgaben genau erfüllen und weder Bursts noch Jitter aufweisen. Somit werden die Anwendungs-Server regelmäßig durch Task-Suspendierungen in einen inaktiven Zustand versetzt. Bei den darauf folgenden Server-Aktivierungen entscheidet die Überprüfung der Aufweck-Regel (CBS-Regel 7) über Kapazitäts-Aufladung und Aktualisierung der jeweiligen Server-Deadline (vgl. Kapitel 4.4.4). Dabei wird der CBS-Scheduler bei Bedarf asynchron aufgerufen (vgl. Listing 4.14). Somit kommt es bei diesem Beispiel nur bei Task-Aktivierungen zu einem Scheduling auf der Applikations-Ebene. Dies wird in der Darstellung in Abbildung 5.14 durch die Aufrufe des asynchronen Event-Handlers des CBS-Schedulers (`AbstractEventHandlerThread[3,pri=27,shared]`) ersichtlich, welcher mit einer Priorität = 27 über die Prioritäts-Bänder der Anwendungen ausgeführt wird. Bei diesem Beispiel wurde das Scheduling auf der Applikations-Ebene also ausschließlich durch diesen Handler betrieben. Dabei wurden A_1 und A_2 entsprechend ihrer aktuellen Server-Deadlines in den Prioritäts-Bändern verschoben, was zu der in Abbildung 5.14 dargestellten Ausführungsreihenfolge ihrer Tasks geführt hat.

Darüber hinaus sind die Event-Handler der applikations-lokalen Task-Scheduler mit den Prioritäten 22 und 25 zu sehen, die bei Task-Suspendierungen von beiden Anwendungen aufgerufen werden, je nachdem in welchem Prioritäts-Band sich die jeweilige Anwendung gerade befindet. Anwendungs-lokales Scheduling findet hier jedoch nicht statt, da beide Anwendungen aus jeweils einer Task bestehen. Dieser Fall wird nun im folgenden Evaluations-Szenario für das Scheduling auf der Applikations-Ebene berücksichtigt, bei dem Anwendungen mehrere Tasks haben können.

5.3.2 Mehrere Tasks pro Anwendung

In diesem Abschnitt wird das Scheduling auf der Applikations- und Task-Ebene im Zusammenspiel evaluiert. Dabei werden wieder zwei Anwendungen innerhalb derselben Plattform ausgeführt, wobei sie nun mehrere Tasks enthalten können. Dadurch wird sowohl die CBS-basierte Planung der Anwendungen als auch das applikations-lokale Task-Scheduling im System aktiv. Wieder wird der Ablauf einer konkreten Konfiguration von Anwendungen und Tasks zunächst theoretisch analysiert und dann mit dem tatsächlichen zur Laufzeit erzeugten Plan im ARTOS-System verglichen.

Die Definitionen der Anwendungen sind in Abbildung 5.15 anhand der generischen Test-Applikation dargestellt. Zur Vereinfachung wurde hier lediglich die erste Anwendung mit zwei

Abbildung 5.15: Konfiguration der Anwendungen mit mehreren Tasks

(in ms)	Periode (P_i)	Kapazität (R_i)	Bandbreite (B_i)
CBS_1	300	170	57 %
CBS_2	500	100	20 %

Tabelle 5.1: Konfiguration der CBS der Anwendungen

Tasks ausgestattet, was jedoch für die Evaluation der Scheduling-Vorgänge auf beiden Ebenen ausreichend ist. Abbildung 5.16 zeigt die Ergebnisse der durch den Scheduler auf der Applikations-Ebene betriebenen lokalen Planbarkeitsanalyse der Anwendungen. Der Wert der Approximationstangente von **App 1** bei der Hyperperiode ihrer Tasks beträgt 850 ms, der von **App 2** ist gleich 100 ms. Die sich nach den Vorgaben aus Kapitel 3.2.2 ergebenden Kapazitäten, Perioden und Bandbreiten der Anwendungs-Server sind in Tabelle 5.1 aufgeführt. Wie man aus den letzten beiden Darstellungen ableiten kann, ist das System sowohl auf der Task-Ebene, wie auch auf der Anwendungs-Ebene mit einer Gesamtauslastung von 77 % planbar. Abbildung 5.18 zeigt den theoretischen Ablauf der Applikations-Tasks innerhalb einer Hyperperiode ihrer CBS sowie den Kapazitätsverlauf des Servers der ersten Anwendung. Aus Übersichtlichkeitsgründen wurde hier die Zeitskala t und die Kapazität c_s des Servers von **App 1** in 100-ten von Millisekunden skaliert. Der Kapazitäts-Verlauf des zweiten Servers ist für diese Evaluation uninteressant, weil seine Ressourcen auf der Task-Ebene nicht zwischen mehreren Aktivitäten aufgeteilt werden müssen. Die Tabelle in Abbildung 5.18 erläutert die Scheduling-Aktivitäten und -Entscheidungen zu unterschiedlichen Zeitpunkten.

Abbildung 5.16: Kapazitätsberechnung der Applikations-Server

Abbildung 5.17: Systemzustand

Wie man im Kapazitäts-Verlauf des Servers von App 1 in Abbildung 5.18 erkennen kann, kommen hier sowohl Frist-Verlängerungen aufgrund Kapazitäts-Entleerungen (z.B. bei $t = 1.7, 5.4, 10.7$) als auch Deaktivierungen (z.B. $t = 4.0, 7.0, 11.0, 13.0$) und Aktivierungen des Servers mit (z.B. $t = 0, 9, 15$) und ohne (z.B. $t = 5.0, 12.0$) vollständige Kapazitäts-Aufladung vor. Wie im Kapitel 4.4.4 bereits erklärt wurde, finden an diesen Stellen die Scheduling-Vorgänge auf der Applikations-Ebene statt. Die Intervalle in denen die Applikationen einzeln aktiv sind, sind anhand ihrer Task-Aktivierungen ersichtlich. So wird z.B. im Zeitpunkt $t = 1.7$ App 1 durch App 2 aufgrund der Verlängerung ihrer Deadline kurzfristig verdrängt. Zum Zeitpunkt $t = 5.0$ werden wiederum die Server von App 1 und App 2 gleichzeitig aktiviert, jedoch wird App 1 nach Anwendung der CBS-Regel 7 aufgrund ihrer kürzeren Deadline zuerst ausgeführt. Wie man in Abbildung 5.18 erkennen kann, ist der Plan gültig und alle Tasks erreichen ihre Fristen rechtzeitig.

Abbildung 5.18: Theoretischer Ablauf des Test-Szenarios

Abbildung 5.17 zeigt den beim Experiment tatsächlichen resultierenden Systemzustand nach ca. 30 Hyperperioden beider Anwendungs-CBS. Die Tasks sind bereits approximiert und die berechneten Bandbreiten und somit die Auslastung des Systems entsprechen den theoretischen Annahmen in Tabelle 5.1. Das System ist planbar und es treten soweit keine Verletzungen der Task-Fristen auf. Betrachten wir nun noch die tatsächlichen Task-Aktivierungen im System bzw. innerhalb der JamaicaVM.

Abbildung 5.19 zeigt die mit dem JamaicaTrace-Tool ermittelten Task-Aktivierungen. Hierbei wird der Ablauf innerhalb der gemeinsamen Hyperperiode aller Tasks in Betracht gezogen, da sich dieser bei gleichbleibenden Schätzungen nicht ändern sollte. Wie man in Abbildung 5.19 sieht, entspricht die zeitliche Abfolge der Task-Aktivierungen im ARTOS-System exakt dem theoretisch errechneten Plan in Abbildung 5.18.

Zum Zeitpunkt $t = 5\text{s}, 860\text{ms}$ werden beide Anwendungen mit ihren Tasks synchron aktiviert. Zunächst bekommen die Tasks von **App 1** aufgrund der kürzeren absoluten Frist ihres CBS Rechenzeit. Dabei wird **Task 2** mit ihrer kleineren Deadline durch den applikations-lokalen Task-Scheduler als erste bis zu ihrer Suspendierung ausgeführt³. Bei ca. $t = 5\text{s}, 960\text{ms}$ wird **Task 3** bereit, ihre Applikation wird jedoch vom CBS-Scheduling immer noch im niedrigeren

³Die `timed wait` Ereignisse im Trace markieren die Task-Suspendierungen in `waitForNextPeriod()`

Abbildung 5.19: Tatsächlicher Ablauf

Prioritäts-Band festgehalten, weswegen sie noch nicht ausführen darf. Ca. 190 ms nach der Aktivierung von **App 1** ist ihre Server-Kapazität von 173 ms (siehe System-Status in Abbildung 5.17) nun aufgebraucht, während **Task 1** noch nicht das Ende ihrer Periode erreicht hat. Hier wird das CBS-Scheduling aktiv und sorgt für eine Verdrängung von **App 1** und eine Aktivierung von **App 2**. **Task 3** bekommt nun die CPU für ca. 100 ms bis zur Beendigung ihrer Periode. Der CBS-Scheduler priorisiert danach wieder die erste Applikation und **Task 1** kann erfolgreich ihre Periode beenden. Die Aktivierung setzt sich dann analog zum theoretischen Plan in Abbildung 5.18 fort.

Interessant zu beobachten sind die Aufrufe des CBS-Schedulers über seinen asynchronen Event-Handler (hier mit der Priorität 27) bei jeder Task-Aktivierung. Diese wurden mit den im Kapitel 4.4.4 vorgestellten Mechanismen bei Server-Aktivierungen (vgl. Seite 152) anhand der `GLOBAL_SCHEDULING`-Priorität der Tasks ermöglicht. Der CBS-Scheduler entscheidet dann bei jeder Task-Aktivierung über die Priorisierung der Anwendungen. Zum Zeitpunkt $t = 7s, 360ms$ werden **Task 1** und **Task 3** gleichzeitig innerhalb ihrer inaktiven Server aktiviert. Dieses Er-

eignis entspricht dem Zeitpunkt $t = 5.0$ im theoretischen Plan. Die Verzögerung durch die Scheduling-Vorgänge auf der Applikations-Ebene führen jedoch dazu, dass bei ca. $t = 7\text{s}$, 460ms **Task 2** aktiviert wird obwohl **Task 1** noch nicht mit ihrer Periode fertig ist (vgl. $t = 6$ in Abbildung 5.18). Dies führt zu einer temporären Verdrängung von **Task 1** durch **Task 2** aufgrund des applikations-lokalen Scheduling. **Task 1** kann jedoch später ihre Periode erfolgreich beenden, bevor das CBS-Scheduling bei ca. $t = 7\text{s}$, 570ms **App 2** und somit **Task 3** aktiviert (vgl. $t = 7$ in Abbildung 5.18). Leichte Abweichungen vom theoretischen Plan können aufgrund des Zusammenspiels mehrerer Faktoren entstehen, wie z.B.:

- geringfügige Schwankungen in den erzielten Task-Kosten aufgrund von Abweichungen von den approximierten Kosten der Scheduling-Vorgänge auf der Task-Ebene – dadurch sind vorzeitige Beendigungen der Tasks wie auch Kosten-Überläufe möglich
- kleinere JVM-interne Verzögerungen bei der Aktivierung der Überlauf-Händlern der CBS
- Verzögerungen der Task-Ausführungen aufgrund kurzfristigen Verdrängungen durch höher priorisierte Event-Handler innerhalb der JVM (z.B. beim Applikations-Scheduling)
- durch die unterliegende Hardware eingeschränkte Präzision der System-Uhr (RTSJ `RealtimeClock`), was zu leichten Verschiebungen der Task-Aktivierungszeiten führen kann – somit kann auch das tatsächliche Eintreffen der Tasks innerhalb der FIFO-Warteschlangen des Schedulers unterschiedlich von der erwarteten Reihenfolge ausfallen

Trotz allem wird die strikte EDF-basierte Reihenfolge der Server und ihrer gekapselten Tasks durch das Scheduling auf der Applikations- und auf der Task-Ebene eingehalten.

Dieses Evaluations-Szenario zeigt insbesondere, dass solange die applikations-lokale und die system-globale Planbarkeit gewährleistet ist, das ARTOS-System auch bei geringfügigen Abweichungen von den vorgegebenen Kosten einen fehlerfreien Aktivierungsplan für Anwendungen mit mehreren Tasks erzeugt. Somit wurde nachgewiesen, dass auch in einem realen System 1) die Approximationen der Anforderungen der Anwendungen die erwarteten Werte erzeugen 2) darauf basierend die Mechanismen des entwickelten hierarchischen Schedulers auf der Task- und Applikations-Ebene im Zusammenspiel einen validen und dem theoretischen entsprechenden Plan erzeugen, 3) dass das System trotz Abweichungen die Kapazitäten der Anwendungen einhalten und ihre Anforderungen auf der Task-Ebene erfüllen kann und 4) dass die entwickelten RTSJ-Erweiterungen für das kapazitäts-basierte Scheduling von Anwendungen innerhalb CBS korrekt funktionieren.

5.4 Evaluation der Adaption-Mechanismen

Bei diesem Evaluations-Szenario werden die automatischen Adaption-Mechanismen von AR-TOS getestet. Dabei werden zunächst zwei Applikationen mit mehreren Modi parallel gestartet und das System in einen stabilen Zustand gebracht. Nach einer bestimmten Zeit wird ein Anstieg im Ressourcenverbrauch einer Applikation emuliert, sodass die Ressourcenkapazität des Systems überschritten wird. Somit wird eine automatische Rekonfiguration erzwungen, bei der eine der Anwendungen in einen niedrigeren Modus degradiert werden muss, um den Engpass zu umgehen. Die Entscheidung welche Anwendung degradiert werden soll, wird durch eine unterschiedliche Priorisierung der Anwendungen beeinflusst. Bei diesem Szenario wird die burstende Anwendung benachteiligt, um die Auswirkungen der Priorisierung zu testen. Dieses Szenario gleicht also konzeptionell dem Fall, bei dem eine neue Anwendung mit einer niedrigeren Priorität im Framework gestartet wird und einen plötzlichen Anstieg der Ressourcenauslastung bewirkt. Im Fokus der Evaluation liegen insbesondere die folgenden Fragen:

1. Wie schnell wird der Anstieg der Kosten erfasst und der Engpass detektiert?
2. Treten währenddessen und auch bei einer Rekonfiguration der Modi Fristverletzungen auf?
3. Erfüllen die Anwendungsprioritäten die tatsächlichen Benutzerpräferenzen?

Abbildung 5.20: Konfiguration der künstlichen Anwendungen

Die Konfiguration beider Applikationen ist anhand der generischen Test-Anwendung in Abbildung 5.20 dargestellt. Wie man dort sehen kann, wurde für jede Anwendung zur Vereinfachung jeweils nur eine Task mit zwei verschiedenen Modi definiert. Die Anwendungsmodi ergeben sich aus den Definitionen der entsprechenden Task-Modi, ihr Nutzen $U_i(M_{i,j}) = j$ wurde durch die generische Test-Anwendung wieder gleich ihrem Index gesetzt. Der niedrigere Modus einer jeden Task relaxiert die Rechenzeitanforderung des höheren. Dafür werden bei der ersten Anwendung die Kosten der Task verringert und auch ihre Deadline verlängert (vgl. App 1, Task1 in Abbildung 5.20). Bei der zweiten Anwendung wird wiederum die Task-Periode vergrößert und die Deadline ebenfalls verlängert. Der zweite Modus der ersten Anwendung emuliert den geplanten Burst durch einen Anstieg der Kosten nach 60 Task-Perioden (30 Sekunden). Die Dauer des Bursts beträgt ebenfalls 60 Perioden. Betrachten wir zunächst die Rechenzeitanforderungen der unterschiedlichen Anwendungsmodi und die entsprechenden Konfigurationen ihrer CBS zur Laufzeit. Diese werden in den Tabellen 5.2 und 5.3 dargestellt.

(in ms)	Periode (P_1)	Kapazität (R_1)	Bandbreite (B_1)
Modus 1	500	100	20 %
Modus 2	500	240	48 %
Modus 2 (Burst)	500	360	72 %

Tabelle 5.2: Applikation 1

(in ms)	Periode (P_2)	Kapazität (R_2)	Bandbreite (B_2)
Modus 1	400	100	25 %
Modus 2	300	133	44 %

Tabelle 5.3: Applikation 2

Solange also der Burst innerhalb der ersten Anwendung nicht auftritt, sollten beide Anwendungen, nach einer initialen Ausmessung ihrer Kosten, gleichzeitig in ihren höheren Modi betrieben werden können. Das System ist dabei theoretisch zu 92 % ausgelastet und beide Anwendungen sind auf der Task-Ebene lokal planbar.

Die Experimente mit den künstlichen Applikationen bestätigen das erwartete Verhalten. Abbildung 5.21 zeigt den stabilen Zustand eines ARTOS Systems mit der oben beschriebenen Anwendungs-Konfiguration nach ca. 15 Sekunden, also nach der Hälfte der Zeit bis zum Auftreten des ersten Bursts. Die Darstellung bestätigt die theoretischen Annahmen mit einer geringfügigen Abweichung im Bandbreiten-Wert der zweiten Anwendung (42 statt 44 %). Bei genauerer Betrachtung sieht man auch, dass die Server-Kapazität von Anwendung 2 mit 127 statt 133 ms

Abbildung 5.21: Systemzustand nach 15 Sekunden

etwas kleiner ausfällt als die weiter oben theoretisch berechnete. Obwohl die Kosten der Task mit einer geringfügigen Abweichung von 1 ms approximiert sind, reicht die Differenz nicht für eine solche Fehlschätzung der Bandbreite aus. Der Grund für den falschen Bandbreiten-Wert liegt im Zeitpunkt des letzten SB-Updates bei der initialen Ausmessung der Task. Bei einer früheren Schätzung wurde der Sicherheitspuffer mit einem kleineren mittleren Wert der Kosten eingestellt und dieser reicht für die Approximation der Task noch aus. Aus diesem Grund ist die Kostenschätzung der Task in Anwendung 2 bei der Momentaufnahme in Abbildung 5.21 bereits auf den erwarteten Wert gestiegen, während die Server-Kapazität aus der letzten Konfiguration noch gültig ist. Diese Abweichung führt auch zu der etwas geringeren als der angenommenen Auslastung des Systems mit 90 % statt 92 %. Besonders zu hervorheben ist die Abwesenheit von Fristverletzungen in beiden Anwendungen. Das Scheduling auf der Task- und Applikations-Ebene sorgt für eine fehlerfreie Aktivierung der Aktivitäten und das System befindet sich soweit in einem stabilen Zustand.

Abbildung 5.22: Systemzustand nach 50 Sekunden

Abbildung 5.22 zeigt den Systemzustand nach ca. 51 Sekunden. Zu diesem Zeitpunkt hat der Burst der ersten Anwendung bereits stattgefunden und das System befindet sich erneut in einem stabilen Zustand, bei dem die niedriger priorisierte Anwendung 1 degradiert wurde. Anwendung 2 befindet sich immer noch im selben Zustand wie vor dem Engpass. Wie man sieht, konnte die Bandbreiten-Auslastung der burstenden Anwendung 1 bis 61 % approximiert werden, bevor der Engpass vom System detektiert und die Konfiguration entsprechend angepasst wurde. Das System wurde also mit ca. 103,3 % leicht überlastet. Das bedeutet jedoch nicht, dass die Anwendungen unter Überlast-Konditionen gelaufen sind. Die Optimierung der System-Konfiguration wird sofort getriggert, sobald ein Sicherheitspuffer aktualisiert und dadurch ein Engpass detektiert wird. Bei einem stetigen Anstieg der Kosten, müsste das System also noch vor einer tatsächlichen Erschöpfung der Ressourcen reagieren können. Für den Fall eines plötzlichen Anstiegs der Task-Kosten, wie bei diesem Szenario, wird in Abbildung 5.23 der Kosten-Verlauf von App 1, Task1 näher untersucht.

Abbildung 5.23: Kosten-Verlauf von App 1, Task1

Wie im Kapitel 3.2.1 bereits erklärt, werden die Sicherheitspuffer einer Anwendung bei jeder Hyperperiode ihrer Tasks periodisch überprüft. In diesem Szenario ist Task1 die einzige Task von App 1, was zu einer Überprüfung der Puffer am Ende einer jeden Task-Periode führt. Wie man in der Abbildung 5.23 sehen kann, steigen die gemessenen Task-Kosten bei Periode 60 auf einen Wert von 300 ms. Die mittleren Kosten verletzen sofort die aktuelle Schätzung. Dies führt

zu einem Update der oberen Grenze des Sicherheitspuffers auf ca. 235 ms, welche nun für die folgende Periode gültig wird. Die lokale Planbarkeitsanalyse für **App 1** ergibt an dieser Stelle eine neue Bandbreiten-Anforderung von 56 % und das System ist nun mit beiden Anwendungen zu 100 % ausgelastet. Periode 62 ist also die letzte Task-Periode, in der die ursprüngliche Konfiguration der Modi gültig ist. Am Ende von Periode 62 bewirkt der weitere Anstieg der Kosten eine sofortige Umschaltung der Anwendung in Modus 1 und eine Absenkung ihrer Rechenzeitanforderung. Dabei wird auch der letzte berechnete Bandbreiten-Wert für den alten Modus (61 %) gespeichert. Im neuen Modus benötigt die Anwendung eine Bandbreite von nur 20 % und das System ist zu ca. 62 % ausgelastet (siehe Abbildung 5.22).

Wie man in beiden Abbildungen 5.22 und 5.23 sehen kann, sind bei beiden Anwendungen trotz Engpass und Rekonfiguration keine Frist-Verletzungen aufgetreten. Je länger jedoch die Detektion eines Engpasses dauert, desto wahrscheinlicher werden auch temporale Fehler. Wie lange es dauert bis ein Burst vollständig erfasst wird, hängt von der Stärke der Schwankung ab und von der Konfiguration des Glättungs-Faktors der mittleren Task-Kosten (vgl. Kapitel 3.1.1). Die Dauer bis zur Detektion eines Engpasses hängt wiederum von der aktuellen Bandbreiten-Belegung im System und von der Periode, mit welcher die Sicherheitspuffer der Tasks überprüft werden, ab.

Da bei diesem Szenario **App 1** nur eine einzige Task besitzt, wurden die Puffer bei jeder Task-Periode überprüft (vgl. Kapitel 4.4.7). Darüber hinaus war der Glättungs-Faktor $\alpha = 0.125$ mit einer starken Gewichtung der Historie eingestellt. Trotzdem war der abrupte Anstieg der Task-Kosten so stark, dass die Überlastung des Systems nach nur zwei Task-Perioden detektiert und die Anwendung rekonfiguriert werden konnte. Dafür waren auch die Bandbreiten der Anwendungen so konfiguriert, dass beim Burst von **App 1** sofort ein Engpass entsteht. Bei diesem Szenario bestand also für nur eine sehr kurze Zeit von zwei Task-Perioden eine Gefahr für Fristverletzungen. Für eine größere Task-Menge innerhalb der Anwendung kann diese Zeit jedoch länger ausfallen, weil sie bei einer raten-basierten Überprüfung der SBs auch von der Hyperperiode des Tasks abhängt (vgl. Kapitel 4.4.7). Darüber hinaus fielen für die Umschaltung der künstlichen Anwendung keine Kosten an, da ihre neuer Modus direkt eingestellt und aktiviert werden kann. Ein anderes Beispiel für die Erfassung eines Bursts wird im folgenden Kapitel 5.5 vorgestellt und untersucht.

Bei einer gleichen Priorisierung würde die Degradierung der zweiten statt der ersten (burstenden) Anwendung im ersten Modus eine ebenso gültige Konfiguration ergeben. Das System wäre mit 97 % Auslastung immer noch planbar und sogar optimaler ausgelastet, während der generierte Nutzen beider Konfigurations-Varianten gleich ist. Die Priorisierung der Anwendungen hat jedoch eine direkte Auswirkung auf ihren Nutzen (vgl. Kapitel 3.2.3). Mit einer Priorität

von 10 erzeugt die zweite Anwendung schon im ersten Modus einen Nutzen von $U_1 = 20$, welcher durch keinen Modus der ersten Anwendung erreicht werden kann. Die höher priorisierte Anwendung wird also tatsächlich bevorzugt, obwohl die Auslastung des Systems möglicherweise nicht optimal ist. Wie man sieht, ist der direkte Einfluss der Priorisierung auf den Nutzen ausschlaggebend für die Verteilung der Ressourcen.

5.5 Erfassung periodischer Bursts

Wie bereits erklärt, wurden zur Evaluation des Modells künstliche Anwendungen entwickelt, die gezielt Last im System erzeugen und ein bestimmtes Burst- und Jitter-Verhalten aufweisen. Damit soll nun analysiert werden, wie oft und aufgrund welcher Faktoren Rekonfigurationen oder Fristverletzungen im System auftreten. Das Burst- und Jitter-Verhalten der Tasks kann ja mit den zusätzlichen Parametern eingestellt werden:

- C_{nom} , C_{max} : nominale und maximale Kosten der Task (CPU-Zeit in ms)
- B_{prd} , B_{dur} : die Periode und die Dauer von Bursts (in Task-Perioden)
- J_{wnd} : Jitter-Fenster der Kosten in Prozent um die vorgegebenen C_{nom} , C_{max}

Interessant dabei ist speziell das Verhalten des Systems bei periodisch auftretenden Bursts. Um den Rekonfigurations-Overhead zu minimieren, müssen zyklische Umschaltungen der Anwendungsmodi möglichst vermieden werden. Dies war eine der ursprünglichen Anforderungen, welche im Kapitel 1.3 an das System gestellt wurden. Abb. 5.24 zeigt die Aktivierung einer Task τ mit den folgenden Vorgaben:

$$\tau = (T = 300, C_{nom} = 50, C_{max} = 90, D = 200, B_{prd} = 20, B_{dur} = 10, J_{wnd} = 20).$$

Das Ziel dieses Experiments ist es zu ermitteln, wie oft die Kostenschätzungen (**Estimate**) aufgrund des periodischen Burst-Verhaltens der Task aktualisiert werden und wo potentielle Gefahren für Fristverletzungen entstehen.

Die Auswirkung der Parametrisierung der Task mit B_{prd} , B_{dur} und J_{wnd} ist in Abb. 5.24 gut zu beobachten. Die gemessenen Kosten (**Measured**) schwanken zufällig jeweils um $C_{nom} = 50$ und $C_{max} = 90$ in einem Fenster von 20 % für eine Dauer von jeweils 10 Perioden. Nach sechs Aktualisierungen ist der Sicherheitspuffer groß genug, um die realen Task-Kosten mit ihren Schwankungen relativ gut abzudecken. Zu einer weiteren Aktualisierung kommt es nicht, da die Burst-Stärke gleich bleibt und die mittleren Kosten den Sicherheitspuffer nicht mehr verletzen.

Abbildung 5.24: Periodisch burstende Anwendung in ARTOS (aus [15])

Zyklisch schwankende Task-Anforderungen werden also automatisch durch die Sicherheitsspeicher erfasst. Dies ist allerdings abhängig von der Größe des Zeitfensters (σ -window), über das die Standardabweichung der gemessenen Task-Kosten berechnet wird (vgl. Formel 3.17, auf Seite 83). In diesem Fall ist das Fenster groß genug um den Burst so zu erfassen, dass nach einer Zeit keine Aktualisierungen mehr stattfinden. Die Folge ist ein Sicherheitspuffer, welcher abschnittsweise die realen Kosten über- bzw. unterprovisioniert. Die freien Ressourcen können zum Beispiel durch einen Capacity Sharing-Mechanismus (CASH) [39] auf CBS-Ebene anderen Applikationen verfügbar gemacht werden und sorgen damit dafür, dass diese weniger wahrscheinlich in Engpässe geraten. Die unterprovisionierten realen Kosten können ebenfalls durch einen solchen Capacity Sharing-Mechanismus bedient werden.

Ein kleineres σ -Zeitfenster führt wahrscheinlicher zu einer stationären Abtastung von Schwankungen und folglich zu frequenteren SB-Aktualisierungen und Rekonfigurationen. In Abb. 5.24 wurde z.B. ein gleitendes σ -Fenster von $B_{prd}/2$ Perioden-Samples gewählt. Würde dieses Fenster stark verkleinert würden irgendwann periodische Rekonfigurationen auftreten. Mit der Wahl der Fenstergröße wird damit bestimmt, welche Periode bei Bursts vom System ausgeglichen

werden kann und welche nicht. Niederfrequente Bursts benötigen zur Erkennung große Fenster, führen aber auch zu stärkerer Überprovisionierung und damit zu schlechterer Auslastung des Systems. Das Verfahren realisiert also einen High-Pass-Filter für die erkennbare Burst-Frequenz, mit steigender Überprovisionierung bei einem größeren Fenster, jedoch auch einer geringeren Rekonfigurations-Rate. Bei einem adaptiven Verfahren kann das Fenster schrittweise vergrößert werden, solange zyklische Bursts immer noch zu Aktualisierungen der Puffer führen. Damit kann ein optimaler Kompromiss zwischen Rekonfigurations-Wahrscheinlichkeit und Überprovisionierung der Ressourcen eingestellt werden.

Die Werte der Standardabweichung und der Sicherheitspuffer werden nur an bestimmten Aktualisierungspunkten überprüft und neu berechnet und zwar nur dann, wenn die Sicherheitspuffer durch die mittleren Task-Kosten verletzt wurden (vgl. Kap. 3.2.1). Diese Punkte werden in der aktuellen ARTOS-Version durch die Hyperperiode der Tasks innerhalb einer Anwendung bestimmt. Dementsprechend können Zeitabschnitte entstehen, bei denen die mittleren Task-Kosten bis zum nächsten Aktualisierungspunkt über den Wert des Sicherheitspuffers steigen (siehe Abb. 3.10b). Da dort die realen Kosten unter Umständen weit über den Puffer hinausragen, ist die Planbarkeit der Task nicht verifiziert und es können potentiell Fristverletzungen auftreten. Sollten Deadlines tatsächlich verpasst werden, schaltet das System automatisch auf eine Strategie um, bei der die Sicherheitspuffer bei jeder Task-Periode überprüft werden (vgl. Abb. 3.10a). Der Overhead und die potentielle Rekonfigurationswahrscheinlichkeit steigen dabei kurzfristig, das System erreicht jedoch schneller einen stabilen Zustand.

Zu einem solchen Fall kommt es ist in Abb. 5.24 nicht. Die Test-Anwendung besteht hier aus einer einzigen Task, wodurch die CBS-Periode mit der Task-Periode identisch ist. Das heißt, dass die SB bei jeder Task-Periode überprüft und unter Umständen aktualisiert werden. Besteht jedoch eine Anwendung aus mehreren Tasks mit einer relativ großen Hyperperiode, so steigt bei einer ratenbasierten SB-Aktualisierung die Wahrscheinlichkeit für Fristverletzungen.

5.6 Evaluation der Optimierung

In diesem Abschnitt werden die Vorgänge auf der höchsten Ebene des Scheduling-Modells von ARTOS evaluiert. Auf dieser Ebene ist die Optimierung des Systems angesiedelt, die zur Laufzeit globale Entscheidungen über die optimale Konfiguration der Anwendungsmodi anhand ihrer ermittelten Ressourcenverbräuche, ihrem Nutzen und der Prioritäten ihrer Anwendungen trifft. Im Kapitel 3.2.3 wurde bereits das entsprechende Optimierungsproblem und eine rucksack-basierte Lösung mittels dynamischer Programmierung vorgestellt. Der Algorithmus auf Seite 113 findet

nicht nur eine optimale Lösung innerhalb einer vollständigen Iteration, sondern verteilt auch die Ressourcen möglichst gleichmäßig unter den laufenden Anwendungen. Seine Eigenschaft bei gleichem Nutzen möglichst viele Anwendungen in niedrigere Modi zu aktivieren, statt denselben Nutzen durch wenige höhergestufte Anwendungen zu erzielen, wurde bereits im Kapitel 5.1.2 an einem realen Beispiel dargelegt. Darüber hinaus wurde im letzten Kapitel 5.4 gezeigt, dass allein die Anwendungsprioritäten eine Gewichtung der Ressourcen unter den Applikationen erlauben. In diesem Kapitel werden nun anhand der prototypischen Implementierung in ARTOS die Auswirkungen einzelner Faktoren auf die Laufzeit der Optimierung untersucht.

Wie im Kapitel 3.2.3 bereits erklärt wurde, hängt die Komplexität des Algorithmus von einer fest einstellbaren numerischen Auflösung der System-Ressourcen und von zwei dynamischen Faktoren ab – der aktuellen Anzahl der Applikationen $|A_i|$ und deren Betriebsmodi $|\mathcal{M}_i|$. Obwohl das Rucksackproblem bekanntermaßen NP-vollständig ist, kann mit der bereitgestellten Lösung theoretisch eine pseudo-polynomiell begrenzte Laufzeit erzielt werden, wenn die numerischen Werte der Faktoren innerhalb angemessener Wertebereiche gehalten werden. Diese Annahme aus Kapitel 3.2.3 wird hier anhand der konkreten Implementierung im ARTOS-System überprüft. Wird von einem System zusätzlich eine bestimmte Höchstdauer für die Berechnung einer optimalen Lösung gefordert, bedeutet diese effektiv eine Begrenzung des Wertebereichs eines jeden dynamischen Faktors – also eine Obergrenze der Anzahl unterstützter Applikationen und Modi. Die Ressourcenauflösung des Systems ist zwar fest einstellbar, muss jedoch auch sorgfältig gewählt werden, weil sie die Ausführungszeit der Lösung mit einem konstanten Faktor bestimmt. Das Ziel dieser Evaluation ist also nicht nur eine Performanz-Abschätzung innerhalb einer konkreten Plattform, sondern auch ein Beispiel-Verfahren, mit dem die Performanz-Grenzen unterschiedlicher Plattform-Konfigurationen und Anforderungen bestimmt werden können.

Zur Erinnerung – bei der Lösung des Optimierungsproblems werden zwei Matrizen $u(k, r)$ und $J(k, r)$ benötigt, deren Dimensionen durch die Anzahl der Applikationen und durch die Ressourcenauflösung des System bestimmt werden (vgl. Kapitel 3.2.3). Der Algorithmus iteriert über alle k Anwendungen und diskreten Ressourcen-Werte $r \in \{0, 1, \dots, R\}$, wobei die Berechnungskomplexität eines jeden Schritts von der Anzahl der Anwendungsmodi $|\mathcal{M}_i|$ abhängt. Aufgrund der Offenheit des Systems hat das ARTOS-Framework zunächst keine Kontrolle über die Anzahl der bereitgestellten Anwendungen und ihrer Modi. Des Weiteren kann der Wert der totalen Systemressourcen R auf einen beliebigen positiven Integer-Wert fest eingestellt werden. Der Anstieg der Berechnungszeit für unterschiedlich große Werte für R , $|A_i|$ und $|\mathcal{M}_i|$ wird nun im Folgenden evaluiert.

Eine entsprechende Realisierung und Integration der Lösung im ARTOS-Framework wurde im Kapitel 4.4.5 diskutiert. Dabei wurde der zunächst als Pseudocode definierte Optimierungsal-

gorithmus (Algorithm 2) mit C implementiert und innerhalb einer nativen Bibliothek bereitgestellt, die über die JNI Schnittstelle innerhalb ARTOS geladen und aufgerufen wird. In den Experimenten wurden vorberechnete Nutzen-Werte für die Applikationsmodi verwendet, die als Prozentsatz an Qualitätsgewinn eines jeden Modus in Bezug auf den höchsten Modus seiner Anwendung definiert wurden (vgl. Kapitel 3.1, Gleichung 3.3). Diese Werte werden zur Laufzeit gemeinsam an den Optimierungsalgorithmus übergeben. Auf diese Weise sind keine weiteren Aufrufe der Utility-Funktionen $U_i(M_{i,j})$ der Anwendungen während der Berechnung der optimalen Konfiguration notwendig.

Die im Folgenden präsentierten Ergebnisse stammen aus früheren Evaluationen, die unter anderem in [14] veröffentlicht wurden. In diesem Kapitel werden nur die wesentlichen Resultate kurz zusammengefasst. Der dabei als Evaluations-Plattform verwendete Desktop-PC verfügte über einen Core(TM)2 Duo Prozessor mit 2×1.4 GHz getakteten Rechenkernen und einem 64 Bit Arch Linux Betriebssystem mit der Kernel Version 3.0 und Glibc Version 2.14. Die Ergebnisse der Messungen entsprechen den gemittelten Werten aus mehreren hundert Versuchen. Sie zeigen akzeptable Ausführungszeiten der Optimierung für bestimmte Konfigurationen des Systems. Es ist zu erwarten, dass die gemessenen Zeiten mit einem schnelleren Prozessor, wie z.B. innerhalb der in Kapitel 5.1.1 präsentierten Evaluations-Plattform, noch kürzer ausfallen, wodurch mehr Anwendungen und Modi mit einer besseren Ressourcenauflösung durch das System unterstützt werden können.

5.6.1 Variation der Ressourcengrenze R des Systems

Prinzipiell ist jede Ressource in irgendeiner Weise limitiert, ihre numerische Repräsentation und somit auch ihre Grenze kann jedoch beliebig gewählt werden. Bei ARTOS entspricht der Wert von R einer 100 %-igen Auslastung der verfügbaren Ressourcenbandbreite. Damit gibt R auch die entsprechende Auflösung vor, mit der die Ressourcenverbräuche der Anwendungen und ihre entsprechenden Bandbreiten erfasst werden. Ein höherer Wert hat eine präzisere Approximation der Verbräuche zur Folge, führt jedoch zu einer höheren Komplexität bei der Optimierung der Modi. Der tatsächliche Wert von R bestimmt direkt die Größe der für die Optimierung verwendeten Tabelle $u(k, r)$ und somit die Anzahl der Rechenschritte in einer Dimension.

Das erste Test-Szenario beinhaltet eine vordefinierte Konfiguration von 10 Applikationen mit jeweils 5 Modi. Dies ist ein durchaus realistisches, wenn auch ein einfaches, Szenario. Die Applikationen werden mit verschiedenen numerischen Werten für die totalen System-Ressourcen getestet. Für die Ressourcenbandbreiten ihrer individuellen Modi wurden unterschiedliche Werte gewählt und für den jeweiligen Wert von R angepasst – höhere Werte von R setzen höhere

Ressourcenanforderungen voraus. Die getesteten Werte von R liegen im Bereich von 128 und 65536 – der Wertebereich eines 16-Bit großen WORD-Datentyps.

R	time[ms]	memory[kB]
128	0.17	6
256	0.31	12
512	0.62	24
1024	1.18	48
2048	2.32	96
4096	4.71	192
8192	9.31	384
16384	18.82	768
32768	38.02	1536
65536	74.67	3072

Abbildung 5.25: Variation der Werte von R (aus [14])

Die Tabelle in Abbildung 5.25 zeigt die Ergebnisse der Evaluation der nativen C-Implementierung. Der Speicherbedarf beinhaltet weder die Applikationen selbst, noch ihre Ressourcenanforderungen oder andere Daten; es werden nur die Allokationen für die Datenstrukturen berücksichtigt, welche für den Algorithmus notwendig sind. Eine Verdopplung des Werts von R resultiert annähernd in einer doppelten Ausführungszeit und den zweifachen Speicherbedarf. Dieser lineare Anstieg ist durchaus plausibel, nachdem eine Verdopplung von R eine Verdopplung der Größe von $u(k, r)$ zufolge hat und somit die zweifache Menge an Werten berechnet und gespeichert werden müssen.

Kleinere Datenstrukturen für den Algorithmus scheinen also eine effiziente Strategie zur Optimierung seiner Laufzeit zu sein. Ein größeres R führt theoretisch zu einer höheren Präzision, diese steht jedoch in einem Widerspruch zu den schwankenden Ressourcenanforderungen der Anwendungen, die das System daran hindern eine exakte Abschätzung zu erzielen. Wenn zum Beispiel der Ressourcenverbrauch der Applikationen mit bis zu 10 % der Gesamtressourcen variiert, können höchstens nur 5 unterschiedliche Modi im System eingebracht werden. Eine Ressourcenauflösung von $R = 100$ wäre dann adäquat. Dieser Wert wird tatsächlich im aktuellen ARTOS-Prototyp verwendet, da er nicht nur eine sehr kurze Algorithmus-Laufzeit bewirkt, sondern auch die Notwendigkeit einer Skalierung der Anwendungs-Bandbreiten für die gewählte Ressourcenauflösung beseitigt. Diese können direkt in ganzzahlige Prozent-Werte der Gesamtressourcen umgewandelt und gespeichert werden. Für die Approximation der bei der Evaluation betriebenen Anwendungen und Szenarien ist diese Ressourcenauflösung hinreichend gut.

Wie bereits erklärt, stellen die gemessenen Zeiten die reine Ausführungszeit des Algorithmus in

seiner nativen Implementierung dar. In einem laufenden ARTOS System kommen noch zusätzliche Kosten für den entsprechenden JNI-Aufruf und die Übergabe der JVM-internen Parameter an den nativen Teil der Implementierung hinzu. Diese Kosten sind jedoch plattformspezifisch und hängen von der Performanz der JNI-Implementierung innerhalb der verwendeten Echtzeit-JVM ab. Eine Evaluation des Overheads der JNI-Schnittstelle verschiedener Echtzeit-JVMs liegt außerhalb des Rahmens dieser Arbeit. Für eine bessere Abschätzung dieses Overheads innerhalb der Konfiguration mit der JamaicaVM wurden jedoch bei den folgenden Experimenten drei verschiedene Implementierungen der Optimierung getestet: a) eine reine Java-Implementierung, die ausschließlich im Interpreter-Modus der JVM ausgeführt wird, b) eine native Implementierung, die über die JNI-Schnittstelle aufgerufen wird und c) die rein-native Implementierung in C.

5.6.2 Variation der Anzahl an Applikationen $|A_i|$

Ähnlich wie die Ressourcengrenze des Systems beeinflusst auch die Anzahl der Anwendungen die Größe der für den Algorithmus benötigten Datenstrukturen und damit auch seine Laufzeit. Eine mögliche Optimierung des Speicherbedarfs des Algorithmus könnte durch eine Minimierung der Matrix $u(k, r)$ auf nur zwei Zeilen realisiert werden, da für die Berechnung einer Reihe nur die Informationen der vorherigen Reihe notwendig sind (vgl. Kapitel 3.2.3). Für $J(k, r)$ ist jedoch die gesamte Matrix notwendig, damit die optimale Konfiguration nach Terminierung des Algorithmus ermittelt werden kann. Dennoch profitiert die Ausführungszeit nicht von dieser Optimierung, da obwohl der Speicherbedarf minimiert werden kann, der Algorithmus immer noch über alle Anwendungen iteriert. Mit jeder zusätzlichen Applikation kommt eine zusätzliche Menge von Modi mit. Diese erhöht die Komplexität zusätzlich, da jeder dieser Modi ebenfalls in die Berechnung einbezogen werden muss. Das bedeutet, dass in Wirklichkeit die Komplexität mit der mittleren Anzahl an hinzukommenden Modi einer jeden Anwendung wächst, also deutlich stärker als bei einer Erhöhung von R .

Nachdem die Ergebnisse im Kapitel 5.6.1 einen kleinen numerischen Wert für die Grenze der Gesamt-Ressourcen vorschlagen, wurde im folgenden Szenario ein Wert von $R = 256$ verwendet. Die Tests variieren in der Anzahl von 10 bis zu 5120 Applikationen mit jeweils 5 Modi. Die Tabelle in Abbildung 5.26 bestätigt die zuvor getroffenen Annahmen. Eine Verdopplung der Anzahl der Applikationen hat ebenfalls eine Verdopplung der Ausführungszeit und des Speicherbedarfs⁴ einer jeden Implementierung zur Folge. Jedoch steigen die Werte schneller als in den im Kapitel 5.6.1 vorgestellten Messungen für unterschiedliche R an, was aus den Zahlen der nativen

⁴Da bei allen Implementierungs-Versionen die selben Daten-Typen verwendet wurden, fällt für sie der Speicherbedarf in Abbildung 5.26 gleich aus.

applications	time[ms](Java)	time[ms](native)	time[ms](JNI)	memory[kB]
10	1.04	0.33	0.75	13
20	2.00	0.63	1.04	23
40	3.96	1.18	2.48	45
80	7.74	2.32	4.81	87
160	15.44	4.72	9.66	172
320	31.14	9.20	15.14	341
410	40.09	11.70	24.56	436
640	63.29	18.30	37.98	680
810	80.48	23.38	48.39	860
1210	120.44	34.60	72.44	1283
1280	128.02	36.45	76.21	1357
1610	160.84	45.89	95.45	1707
2010	201.06	57.75	119.75	2130
2410	241.86	68.70	144.21	2553
2560	257.06	72.98	150.80	2712
2810	281.70	80.22	168.14	2977
3210	322.26	91.90	191.97	3400
3610	363.34	103.39	215.66	3824
5120	515.33	146.68	302.48	5422

Abbildung 5.26: Variation der Menge an Applikationen (aus [14])

Version ableitbar ist. Die Ergebnisse zeigen außerdem, dass die Kosten der JNI-Aufrufe ungefähr den doppelten Aufwand an Rechenzeit erfordern und dass die reine Java-Implementierung die langsamste ist.

Die Ausführungszeiten der nativen Implementierung in Abbildung 5.26 scheinen für den Bereich von 50..100 Anwendungen immer noch akzeptabel zu sein. Die tatsächliche Anzahl an Applikationen ist jedoch vom jeweiligen Anwendungsszenario abhängig. Wird vom System zum Beispiel für eine zugelassene Menge von höchstens 80 Anwendungen mit jeweils maximal 5 Modi eine kürzere Laufzeit als 2.23 ms vorausgesetzt (vgl. Abb. 5.26), kann nun der Wert von R entsprechend verkleinert werden. Anhand der Zahlen aus den Experimenten im vorherigen Kapitel 5.6.1 lässt sich nun aufgrund des linearen Anstiegs der Laufzeiten ableiten, dass für ein $R = 128$ bei 80 Anwendungen mit jeweils 5 Modi der Algorithmus im Durchschnitt nur 1.36 ms benötigt und somit die Voraussetzungen des Systems erfüllt. Innerhalb einer gängigen Desktop-Umgebung ist jedoch mit einer höheren Anzahl als 20 aktiven Anwendungen eher selten zu rechnen.

5.6.3 Variation der Anzahl an Modi $|\mathcal{M}_i|$ pro Applikation

Die zweite innere `for`-Schleife (in Zeile 7) im Algorithmus 2 auf Seite 113 iteriert über jeden Modus einer Anwendung und ist somit der dritte hauptsächliche Faktor, welcher die Ausführ-

rungszeit der Optimierung beeinflusst. Eine Erhöhung der durchschnittlichen Anzahl an Modi um 1 bedeutet einen zusätzlichen Berechnungsschritt für jede Position in $u(k, r)$ und $J(k, r)$.

Es ist nicht sinnvoll, dass die Anzahl an Modi über den Integer-Wert von R steigt, da ihre Ressourcenanforderungen nicht die Ressourcengrenze des Systems übersteigen können. Aus diesem Grund ist es bei den Tests notwendig, dass der Wert für R erhöht wird. Die Tabelle in Abbildung 5.27 zeigt die Ergebnisse eines Tests für $R = 3000$, 10 Applikationen und Anzahl an Modi zwischen 50 und 1950. Schon für 50 Modi pro Applikation steigt die Ausführungszeit der Berechnung enorm an. Dies ist nicht überraschend, nachdem schon im Kapitel 5.6.1 gezeigt wurde, dass allein die Erhöhung von R längere Ausführungszeiten bewirkt. Ein Anstieg der durchschnittlichen Anzahl an Modi multipliziert die Ausführungszeiten mit der Anzahl an Applikationen. Die Ergebnisse dieses Tests liegen weit über den akzeptablen Werten für ein Echtzeit-System.

modes	time[ms](Java)	time[ms](native)	time[ms](JNI)	memory[kB]
50	60.2	17.6	38.9	141
150	171.3	49.6	107.4	141
250	276.6	76.8	170.4	141
450	467.9	125.3	290.2	141
550	549.6	146.3	345.3	141
650	634.3	168.8	401.0	141
850	792.9	206.4	498.7	141
950	862.2	224.9	546.2	141
1050	937.3	245.5	589.9	141
1350	1124.7	290.0	709.6	141
1550	1225.3	316.9	779.4	141
1750	1323.5	341.2	838.3	141
1950	1404.1	371.0	894.6	141

Abbildung 5.27: Variation der Menge an Modi pro Applikation (1) (aus [14])

Dennoch ist eine hohe Anzahl an Modi pro Applikation nicht zu erwarten. Auch wenn die Ressourcenanforderungen der individuellen Modi nur geringfügig schwanken und somit prinzipiell mehr Anwendungen eingebracht werden können, sind mehr als 20 unterschiedliche Modi pro Applikation nicht denkbar. Ein anderer Test mit einer Ressourcengrenze von $R = 256$, 10 Applikationen und bis zu 50 Modi pro Applikation zeigt in Abbildung 5.28 deutlich kürzere Ausführungszeiten. Anhand eines solchen Experiments kann zum Beispiel die zulässige Menge an Modi pro Applikation für eine maximale Anzahl an Anwendungen in einem System definiert werden.

In den beiden Abbildungen 5.27 und 5.28 ist zu erkennen, dass der benötigte Speicher für den Algorithmus konstant bleibt. Der Algorithmus muss für die zusätzlichen Modi keinen zusätzlichen

modes	time[ms](Java)	time[ms](native)	time[ms](JNI)	memory[kB]
1	0.19	0.08	0.14	12
5	0.62	0.20	0.43	12
9	0.99	0.32	0.71	12
13	1.36	0.44	0.96	12
17	1.79	0.56	1.17	12
21	2.10	0.63	1.34	12
25	2.48	0.71	1.57	12
29	2.78	0.79	1.81	12
33	3.10	0.91	1.99	12
37	3.43	0.95	2.15	12
41	3.72	1.07	2.39	12
45	4.05	1.12	2.58	12
49	4.34	1.18	2.79	12

Abbildung 5.28: Variation der Menge an Modi pro Applikation (2) (aus [14])

Speicher allozieren, da alle Informationen bereits als Eingabedaten der Optimierung verfügbar vorliegen. Die Datenstrukturen für die Berechnungen wachsen nur, wenn die Ressourcengrenze oder die Anzahl der Applikationen erhöht wird.

5.6.4 Aggregation der Faktoren

Wie die Ergebnisse der vorherigen Kapiteln zeigen skaliert die Ausführungsdauer der Berechnung einer optimalen Modi-Konfiguration linear in jeder Dimension. Der Grund dafür ist, dass bei jedem Test nur eine der drei relevanten Variablen modifiziert wurde. In diesem Abschnitt werden nun Korrelationen zwischen den Variablen untersucht. Diese müssten sich bei gleichzeitiger Modifikation mehrerer Faktoren durch Unregelmäßigkeiten in den gemessenen Ausführungszeiten der Optimierung bemerkbar machen. Sollten jedoch alle Variablen voneinander unabhängig sein, dann müsste ein kumulativer linearer Anstieg der Zeiten nachweisbar sein.

	R	Apps	Modes	Runtime [ms]	Mem [kB]
x2 {	128	10	5	0,17	6
	256	20	5	0,6	23
				x4	x4

Abbildung 5.29: Variation der Menge an Applikationen und der Ressourcengrenze (1)

Die Tabelle in Abbildung 5.29 zeigt einen kleinen Auszug aus den Ergebnissen der Evaluation der nativen Implementierung, für den gleichzeitigen Anstieg der Werte innerhalb zwei Dimensionen, R und $|A_i|$. Wie man dort sieht, bewirkt die Verdopplung der Werte in den zwei Dimensionen eine Vervierfachung der Laufzeit und des Speicherbedarfs. Die Ressourcengrenze des Systems R und die Menge der Applikationen $|A_i|$ sind also voneinander unabhängig und tragen separat mit ihrem linear wachsenden Anteil zur gesamten Rechenzeit bei. Abbildung 5.30 zeigt den weiteren Anstieg der Ausführungszeiten aller drei Implementierungs-Versionen.

Abbildung 5.30: Variation der Menge an Applikationen und der Ressourcengrenze (2) (aus [14])

Auf ähnliche Weise wurden ebenfalls Korrelationen zwischen den anderen Variablen R und $|\mathcal{M}_i|$ sowie A_i und $|\mathcal{M}_i|$ untersucht. Auf die vollständige Darstellung der Ergebnisse wird hier jedoch verzichtet, diese können in [14] weiter verfolgt werden. Für alle Variablen wurde eine Unabhängigkeit nachgewiesen. Dieser Abschnitt zeigt also, dass der entwickelte Algorithmus eine einfache System-Analyse durch lineare Abschätzungen der Laufzeiten ermöglicht. Dabei können für eine Plattform-Konfiguration exakte Performanz-Grenzen bestimmt und Anforderungen an die Laufzeit der Optimierung umgesetzt werden.

5.7 Evaluation des Qualitätsgewinns anhand einer VoD-Anwendung

In diesem letzten Evaluations-Szenario wird der durch die in ARTOS unterstützten Degradierungs-Mechanismen von Applikationen gewonnene Nutzen anhand eines realen Anwendungsbeispiels evaluiert. Ähnlich wie bei dem in Kapitel 2.2 bereits vorgestellten Ansatz der

Imprecise Computations wird hier die Grundannahme getroffen, dass ein rechtzeitiges Resultat von geringerer Qualität einen höheren Nutzen erzeugt, als ein verspätetes Ergebnis mit der bestmöglichen Qualität. Somit wird ein gültiger Ablaufplan ohne Fristverletzungen mit einem höheren Nutzen bewertet als einer, bei dem temporale Fehler auftreten. Diese Evaluation baut auf einer realen Video-on-Demand (VoD) Anwendung auf, die speziell für das ARTOS System entwickelt wurde und unterschiedliche Betriebsmodi und somit Qualitäten eines gestreamten Videos unterstützt. Der erzeugte Nutzen durch die Anwendung wird also auch an den tatsächlichen Qualitätseinstellungen des Videos in den unterschiedlichen Betriebsmodi festgemacht. Mit diesem Szenario wird das ursprüngliche Beispiel aus Kapitel 1.2 nachgebildet, welches als Motivation der Arbeit diene.

Die VoD-Anwendung besteht aus einem Sender (bzw. Server) und einem Empfänger (bzw. Client), die auf unterschiedlichen Rechnersystemen ausgeführt werden. Der Sender encodiert ein vom Empfänger angefordertes Video und überträgt dieses über eine Netzwerkverbindung zum Client. Dabei unterstützt der Server unterschiedliche Qualitätsmodi, in die er dynamisch zur Laufzeit umgeschaltet werden kann. In jedem Modus wird das Video-Signal mit einem unterschiedlichen Bild-Format und Qualitätseinstellungen encodiert und zum Client übertragen. Dort wird es dann entsprechend decodiert und innerhalb eines Fensters wiedergegeben. Ein Benutzer ist somit in der Lage die Qualitätsentscheidungen des Frameworks „live“ mitzuverfolgen und die Anwendung innerhalb seiner Plattform entsprechend zu priorisieren.

Je nach Qualitätseinstellungen und Format des übertragenen Videos benötigt der Client eine unterschiedliche Menge an Ressourcen, um die empfangenen Video-Daten zu decodieren und darzustellen. Die Beziehung zwischen Qualität und Ressourcenverbrauch wurde bereits im Rahmen des Kapitels 2.4, unter anderem am Beispiel der Codierung und Decodierung von Video-Signalen, thematisiert. In diesem Evaluations-Szenario werden nur die CPU-Ressourcen des Clients durch ARTOS verwaltet. Es wird außerdem davon ausgegangen, dass der Sender über genügend Ressourcen verfügt, um das Video in allen Qualitätsmodi einwandfrei zu encodieren und zu versenden. Darüber hinaus wird das Netzwerk als möglicher Engpass abstrahiert, indem sicher gestellt wird, dass es genügend Bandbreite und eine minimale Übertragungs-Latenz besitzt, damit die Videodaten in allen Modi rechtzeitig übertragen werden können. Qualitätsanpassungen sollen nur aufgrund der Auslastung der CPU-Ressourcen beim Client stattfinden.

Bei dem Beispiel-Szenario im Kapitel 1.2 erzeugt ein eingehender Anruf innerhalb einer Internettelefonie (VoIP)-Anwendung einen Engpass im System und der Benutzer wird aufgefordert entweder die VoD- oder die VoIP-Anwendung zu priorisieren, um den Engpass zu umgehen. Dabei wird die Anwendung mit der niedrigeren Priorität entsprechend in einen niedrigeren Qualitätsmodus degradiert. In diesem Evaluations-Szenario wird die VoIP-Anwendung durch eine

künstliche ARTOS-Anwendung emuliert, die einen bestimmten Ressourcenverbrauch und somit einen Engpass gezielt erzeugen kann. Wird der Engpass durch das ARTOS-Framework detektiert, dann soll die VoD-Anwendung in einen niedrigeren Modus geschaltet und der Sender über den Modus-Wechsel benachrichtigt werden. Die automatischen Anpassungs-Mechanismen und die Dauer bis zur Detektion eines Engpasses im ARTOS-System wurden bereits im Kapitel 5.4 evaluiert und diskutiert.

Das Ziel dieses Evaluations-Szenarios ist eine quantitative Einschätzung der erbrachten Qualität beim Client. Dafür wird die beschriebene Engpass-Situation mit und ohne die Qualitätsanpassungs-Mechanismen von ARTOS nachgebildet. Um die Auswirkungen des Engpasses genau zu evaluieren, werden die Aktivitäten innerhalb der VoD-Anwendung als periodische Echtzeit-Tasks mit festen Fristen modelliert und das wiedergegebene Video beim Client zwischengespeichert.

Wird die Degradierung der Anwendungen in ARTOS deaktiviert, dann ist zu erwarten, dass der Engpass zu permanenten Anforderungsverletzungen innerhalb beider Anwendungen führt. Diese äußern sich speziell bei der VoD-Anwendung entweder in einem Verlust von zu decodierenden Daten oder als Verzögerung der Wiedergabe. Dabei entstehen somit Lücken im zwischengespeicherten Video-Signal beim Client. Das empfangene Video wird dann frameweise mit dem Original-Video auf dem Server verglichen, wobei das Spitzen-Signal-Rausch-Verhältnis (PSNR) (vgl. Kapitel 2.4.2) zwischen beiden berechnet wird. Für die Stellen, an denen kein Signal empfangen und rechtzeitig dargestellt werden konnte ist ein sehr niedriger PSNR-Wert zu erwarten, während die übrigen Frames identisch sein sollten und somit einen hohen PSNR-Wert ergeben. Dadurch lässt sich unter anderem ein mittlerer PSNR-Wert für die gesamte übertragene Video-Sequenz errechnen.

In einem zweiten Test werden die Qualitätsanpassungs-Mechanismen von ARTOS scharf gestellt. Nach der entsprechenden Degradierung der VoD-Anwendung beim Engpass, speichert der Client Frames mit einer niedrigeren Auflösung und Bildqualität. Außerdem werden die Anforderungen der Tasks durch eine niedrigere Framerate und somit längere Periode bzw. Frist im niedrigeren Modus relaxiert. Die Degradierung der Anwendung soll den Engpass im System beheben. Da hier längerfristig kein Bild-Material aufgrund von Anforderungsverletzungen verloren geht, sondern die durchschnittliche Qualität der Frames heruntergestuft wird, ist insgesamt mit einem höheren durchschnittlichen PSNR-Wert zwischen dem Original und dem empfangenen Video zu rechnen.

5.7.1 VoD-Anwendung

In diesem Abschnitt wird zunächst der Aufbau und die Funktionsweise der VoD-Anwendung sowie die Konfiguration des Systems bei den Test-Szenarien näher beschrieben. Anschließend wird das Verhalten der VoD-Anwendung zur Laufzeit analysiert und dabei die Vorgaben für den Ressourcenverbrauch der künstlichen Applikation innerhalb der Testszenarien abgeleitet.

Aufbau und Funktionsweise

Abbildung 5.31: Architektur der VoD-Applikation

Abbildung 5.31 veranschaulicht grob die Architektur der VoD-Anwendung. Ein Sender encodiert Video-Daten und verschickt diese über ein Netzwerk zu einem Empfänger. Die empfangenen Video-Daten werden zunächst zwischengepuffert und parallel von einem Darsteller decodiert und visualisiert. Alle Aktivitäten, sowohl auf der Sender- als auch auf der Empfänger-Seite, werden als periodische Tasks mit Echtzeitanforderungen modelliert. Auf der Sender-Seite reicht der Einsatz eines einfachen RTSJ `RealtimeThreads`. Auf der Client-Seite werden der Empfänger und der Darsteller als ARTOS-Tasks mit mehreren Modi realisiert und somit in den Scheduling-Entscheidungen des Frameworks integriert. Obwohl der Sender ebenfalls unterschiedliche Qualitätsstufen unterstützt, werden diese vom Client aus durch Benachrichtigungen über einen Kommunikationskanal gesteuert. Daraufhin passt der Sender die Qualität des versendeten Video-Signals dynamisch an. Die Video-Daten werden sequentiell aus einer Datei ausgelesen, in das entsprechende Format transcodiert und mit der gewünschten Rate frameweise über eine

TCP-Verbindung zum Client versendet. TCP wurde hier als Transport-Protokoll gewählt, um Netzwerk-Fehler auszuschließen und den Fokus auf die CPU-Ressourcen beim Client zu legen.

Die Benachrichtigungen über Modi-Wechsel über den Rückkanal zum Sender wurden ebenfalls mit einer TCP-Verbindung kommuniziert. Im Falle von einer RTP-basierten Übertragung der Video-Daten könnte ein Modus-Wechsel auch über eine Erweiterung des RTCP-Protokolls realisiert werden (vgl. Kapitel 2.3.2). Dabei unterstützt RTCP unterschiedliche Mechanismen für eine Erweiterung seiner bestehenden Profile mit neuen Metriken [133]. Im Rahmen dieser Arbeit war jedoch eine einfache TCP-Verbindung zum Server völlig ausreichend.

Die Anwendung unterstützt drei Qualitätsmodi, welche sich als Betriebsmodi der Empfänger- und Darsteller-Tasks äußern. Die entsprechenden Qualitätsanpassungen des übertragenen Videos werden durch unterschiedliche Konfigurationen der Bildauflösung, Framerate und der Bitrate realisiert (siehe Kapitel 2.4). Alle Tasks, sowohl beim Client als auch beim Server, werden für eine vorgegebene Framerate durch eine entsprechende Echtzeit-Periode synchronisiert. Somit wird sichergestellt, dass die vom Darsteller-Thread beim Client benötigten Video-Daten rechtzeitig ausgeliefert und seine Anforderungen erfüllt werden können. Die Tabelle in Abbildung 5.32 zeigt die Konfiguration der Modi der Anwendung.

Modus	Breite	Höhe	Framerate	Bitrate
1	480 Pixel	270 Pixel	10 fps	50.000 kbps
2	1280 Pixel	720 Pixel	20 fps	8.000.000 kbps
3	1920 Pixel	1080 Pixel	30 fps	40.000.000 kbps

Abbildung 5.32: Modi der VoD-Applikation

Die Qualität des Videos kann in den unterschiedlichen Modi von einer geringen bis zu einer Full HD-Auflösung variiert werden. Die unterschiedlichen Frameraten resultieren als entsprechende Task-Perioden mit 100, 50 und 34 Millisekunden. Die unterschiedlichen Bitraten haben eine Auswirkung auf die Quantisierungsgenauigkeit beim Encodieren und somit auf die zu berechnende Datenmenge beim Dekoder (vgl. Kapitel 2.4). Es ist zu erwarten, dass bei einem niedrigeren Modus ein geringerer Ressourcenverbrauch beim Client deutlich zu erkennen ist, während die Qualitätsunterschiede im wiedergegebenen Video direkt zu sehen sind. Um Schwankungen im Ressourcenverbrauch durch Bewegungen in der übertragenen Video-Szene zu minimieren, wurden ausschliesslich I-Frames codiert und versendet. Das Auslassen von B- und P-Frames hat auch den Vorteil, dass die Auswirkung einer Verletzung der zeitlichen Anforderungen der Anwendung nur auf die Qualität des gerade verarbeiteten Frames beschränkt wird.

Der Sender besteht aus einer Kommunikations- und einer Transcoder-Komponente. Der Transcoder encodiert das angeforderte Video frameweise nach den entsprechenden Qualitätsvorgaben mit Hilfe der FFmpeg Bibliothek. Als Codec wurde MPEG-4 Part-2 gewählt, da dieses Format eine variable Framerate unterstützt. Die Kommunikations-Komponente ist zuständig für das Versenden der encodierten Daten und für das Empfangen von Benachrichtigungen über Modi-Wechsel vom Client. Der Sender wurde mit Echtzeit-Java entwickelt und innerhalb der JamaicaVM ausgeführt. Die Aufrufe der nativen FFmpeg Bibliothek wurden durch JNI mittels einer für dieses Szenario eigens entwickelten nativen Transcoding-Bibliothek realisiert.

Der Client wurde wiederum als ARTOS-Anwendung auf einem dedizierten Rechner ausgeführt. Das ARTOS-System misst die Kosten des Empfänger- und des Darsteller-Threads automatisch und überwacht das Auftreten von Fristverletzungen. Bei einer Fristverletzung wird die Darstellung des entsprechenden Frames abgebrochen und an seiner Stelle ein schwarzes Bild mit dem entsprechenden Format im ausgegebenen Video gespeichert. Auf diese Weise werden die Format-Vorgaben auch bei Fristverletzungen während eines Modus-Wechsels eingehalten. Beim Decodieren ruft der Darsteller-Thread ebenfalls die FFmpeg Bibliothek über JNI auf. Für die Visualisierung des Videos wurde die Simple DirectMedia Layer (SDL) [134] Bibliothek verwendet, welche es ermöglicht, einzelne Frames innerhalb eines nativen X-Window Fensters darzustellen. Die SDL-Funktionen wurden dabei ebenfalls über eine spezielle JNI-Bibliothek aufgerufen.

Bei den Experimenten wurde das Netzwerk soweit wie möglich abgeschottet. Um trotzdem mit Verzögerungen bei der Übertragung der einzelnen Paketen und bei der Benachrichtigung und Verarbeitung der Modi-Wechsel umgehen zu können, werden die Frames auf der Empfänger-Seite vor der Darstellung für jeden Modus separat gepuffert (vgl. Abbildung 5.31). Der Empfänger- und der Darsteller-Thread werden somit voneinander entkoppelt. Letzterer fängt mit dem Decodieren erst dann an, wenn der entsprechende Puffer ausreichend gefüllt ist. Dies hat zur Folge, dass bei einem Modus-Wechsel die bereits gepufferten Frames des letzten aktiven Modus zunächst vollständig abgearbeitet werden müssen, bevor die Einstellungen des Darsteller-Threads verändert werden. Somit wird eine kontinuierliche Wiedergabe des Videos gewährleistet. Dagegen wird ein Modus-Wechsel sofort zum Sender kommuniziert, welcher dann die Qualität entsprechend verändert und den Puffer des neuen Zielmodus befüllt, bis schließlich der Darsteller-Thread umgeschaltet wird.

Die bei der Wiedergabe eines Beispiel-Videos durch ARTOS ermittelten Laufzeitkosten der unterschiedlichen Modi der VoD-Anwendung werden in der Abbildung 5.33 anhand des aktuellen System-Status dargestellt. Wie man dort sehen kann, bleiben die Kosten des Empfänger-Threads mit 3 - 4 ms pro Periode bei allen Modi konstant. Dies ist nachvollziehbar, da die Arbeit des Empfänger-Threads bei allen Modi identisch ist und nur seine Periode entsprechend modifiziert

(a) Modus 3

(b) Modus 2

(c) Modus 1

Abbildung 5.33: Laufzeitkosten der Modi der VoD-Anwendung

wird. Die Kosten des Darstellungs-Threads unterscheiden sich dagegen in den verschiedenen Modi mit 9, 6 und 3 Millisekunden deutlich genug und schwanken mit einer Standardabweichung von $\sigma < 1$ nur geringfügig. Durch die unterschiedlichen Perioden und Fristen (gleich den Perioden) der Tasks ergeben sich unterschiedliche Bandbreiten-Anforderungen der Anwendungsmodi. So benötigt der höchste Modus 3 ca. 42 %, der mittlere Modus 2 ca. 18 % und der niedrigste Modus 1 ca. 10 % der verfügbaren System-Ressourcen.

Definition der künstlichen Test-Anwendung

Mit den ermittelten Kosten der Modi der VoD-Anwendung kann nun der Ressourcenverbrauch der künstlichen ARTOS-Applikation definiert werden, welche bei den Test-Szenarien den eingehenden VoIP-Anruf emuliert und einen Engpass erzeugt. Abbildung 5.34 zeigt die Modellierung der Applikation mit der generischen Test-Anwendung von ARTOS.

Abbildung 5.34: Definition der künstlichen Anwendung für den Engpass

Abbildung 5.35: Systemzustand beim Engpass ohne Degradierung

Da bei den Szenarien nur die VoD-Anwendung degradiert werden soll, besitzt die Test-Anwendung nur einen einzigen Modus. Dabei erzeugt eine Task eine relativ hohe Rechenlast und die Anwendung belegt 70 % der verfügbaren Ressourcenbandbreite. Bei einer gemeinsamen Ausführung mit der VoD-Anwendung im höchsten Modus würde eine System-Auslastung von geschätzt 112 % zu einer Degradierung der VoD-Anwendung im zweiten Modus führen. Betrachten wir als nächstes das Szenario ohne die automatischen Anpassungs-Mechanismen von ARTOS.

5.7.2 VoD-Szenario ohne Degradierungs-Mechanismen

Bei diesem Szenario wurden beide Anwendungen synchron gestartet und gleich priorisiert. Bei der VoD-Anwendung wurde eine Beispiel-Videosequenz mit höchster Qualität übertragen. Um eine automatische Degradierung der VoD-Anwendung zu verhindern, wurde diese in ihrem höchsten Modus befestigt (gepinnt) (vgl. Kapitel 5.1.2). Abbildung 5.35 zeigt den aktuellen Systemzustand nach ca. 46 Sekunden.

Wie man in der Abbildung erkennen kann, leiden die Tasks beider Anwendungen aufgrund der Überbelastung des Systems unter Deadline-Misses. Dabei kommt es bei den kürzeren Perioden und Fristen der Tasks innerhalb der VoD-Anwendung häufiger zu Anforderungsverletzungen als bei der künstlichen Anwendung. Da beim Auftreten von Fristverletzungen die aktuellen Task-Jobs abgebrochen werden, resultieren diese in Schwankungen innerhalb ihrer gemessenen Ausführungszeiten. Dies sorgt für eine stärkere Abweichung von den mittleren Werten der Tasks und somit für größere Sicherheitspuffer und Kostenschätzungen. Daraus resultiert eine höhere Bandbreiten-Auslastung der Anwendungen und des Gesamtsystems als zuvor angenommen. Betrachten wir nun die Kosten der einzelnen Tasks innerhalb der VoD-Anwendung.

Abbildung 5.36 stellt die gemessenen, die mittleren sowie die Schätzungen der Kosten beider Tasks innerhalb des gleichen Zeitabschnitts dar. Die vertikalen roten Linien zeigen die Stellen an, bei denen Fristverletzungen aufgetreten sind. An diesen Stellen wurden, wie bereits erklärt, keine Video-Daten decodiert oder bereitgestellt, sondern stattdessen schwarze Bilder im Ausgangs-Signal gespeichert. Da hier Fristverletzungen auftreten, hat das ARTOS-System bei allen Anwendungs-Tasks eine instantane Aktualisierungs-Strategie ihrer Sicherheitspuffer eingestellt (vgl. Abbildung 3.10 im Kapitel 3.2.1). Wie man sieht, bleiben die Kosten der Empfänger-Task relativ konstant, während die der Darsteller-Task abhängig von der aktuellen Video-Szene schwanken. Bei diesem Experiment hat die Empfänger-Task stärker unter Anforderungsverletzungen gelitten als die Darsteller-Task, was nicht immer so sein muss. Bei einem Engpass breiten sich Fristverletzungen nicht-deterministisch auf die Tasks im System aus, abhängig von ihrer Aktivierungsreihenfolge zum Zeitpunkt des Bottlenecks [12].

Average PSNR	81.28
Maximum PSNR	99.00
Minimum PSNR	4.0
Average of Values < 99 dB	7.6

Tabelle 5.4: PSNR-Ergebnisse beim Engpass-Szenario ohne Degradierung

Bei einem letzten Test wurde der Spitzen-Signal-Rausch-Abstand (PSNR) zwischen dem versendeten Original-Video und dem empfangenen und dargestellten Video beim Client berechnet. Die Ergebnisse sind in der Tabelle 5.4 dargestellt. Der durchschnittliche PSNR-Wert zwischen beiden Videos ist, trotz stellenweisem Verlust der Video-Daten, mit 81.28 dB immer noch relativ hoch. Der maximal gemessene PSNR-Wert von 99.00 dB entspricht einem identischen Bild beim Sender und beim Empfänger. Der minimale gemessene Wert von 4.0 dB ergibt sich aus

(a) Empfänger-Task

(b) Darsteller-Task

Abbildung 5.36: Task-Kosten beim Engpass-Szenario ohne Degradierung

Abbildung 5.37: Systemzustand beim Engpass mit Degradierung

zufälligen Überlappungen von schwarzen Abschnitten im Original-Bild mit einem künstlich hinzugefügten schwarzen Frame beim Empfänger. Der mittlere Wert aller Signal-Abstände kleiner als 99 ist wiederum ein Maß für die durchschnittliche Qualität der Frames, die nicht mit dem Ursprungs-Signal identisch sind. Da bei diesem Szenario die Video-Daten bei Anforderungsverletzungen verworfen werden, ergibt sich hierfür ein relativ schlechter Übereinstimmungs-Wert von 7.6 dB.

5.7.3 VoD-Szenario mit Degradierungs-Mechanismen

Bei diesem letzten Evaluations-Szenario wird dem ARTOS-System die Möglichkeit gewährt, die VoD-Anwendung beim künstlich erzeugten Engpass automatisch zu degradieren. Dafür wird die Anwendung in keinem ihrer Modi gepinnt. Erneut werden alle Applikationen im System synchron gestartet, wobei mit der VoD-Anwendung dieselbe Video-Sequenz wie beim letzten Evaluations-Szenario im Kapitel 5.7.2 abgespielt wurde. Abbildung 5.37 zeigt den aktuellen Systemzustand nach einer Ausführungszeit von ca. 46 Sekunden.

Wie man sieht, wurde die VoD-Anwendung in ihrem mittleren **Modus 2** aktiviert und bis auf bei der Empfänger-Task kommen soweit keine Fristverletzungen vor. Der Grund für letzteres wird weiter unten näher erläutert. Beide Anwendungen wurden in ihrem aktuellen Zustand korrekt approximiert und die aktuelle System-Auslastung beträgt ca. 90 %. Darüber hinaus schwanken die Task-Kosten nicht mehr so stark wie beim letzten Szenario, da sie nun bei jeder Periode ohne Fristverletzungen vollständig approximiert werden können. Abbildung 5.37 zeigt den Zustand des Systems nach der letzten Rekonfiguration an. Um zu sehen, ob und wann Fristverletzungen vor

der Modus-Umschaltung aufgetreten sind, werden nun erneut die Plots der ermittelten Kosten beider Tasks näher betrachtet.

Abbildung 5.38 zeigt den Verlauf der Kosten beider Tasks mit Degradierung der VoD-Anwendung. Erneut ist der Anstieg der Kosten der Darsteller-Task aufgrund der aktuellen Video-Szene um 100 Perioden herum auffällig, welcher zu einer geschätzten Bandbreite von ca. 35 % im **Modus 3** und somit zu einem Engpass im System geführt hat (zu Erinnerung, die künstliche Anwendung belegt eine Bandbreite von ca. 70 %, vgl. Abbildung 5.37). Zu diesem Zeitpunkt wird auch der Bottleneck von ARTOS detektiert, was zu einer sofortigen Umschaltung der Anwendung im **Modus 2** geführt hat. Dabei wird zunächst nur der Modus der Empfänger-Task gewechselt. Die tatsächliche Umschaltung der Darsteller-Task findet zeitverzögert bei ca. 130 Perioden statt, da die Task zunächst die bereits gepufferten Frames für **Modus 3** vollständig abarbeiten muss. Erst danach wechselt die Task in den neuen Modus und ihre Kostenschätzung etabliert sich bei relativ geringen Schwankungen mit einem kleinen Puffer um den mittleren Wert von ca. 6 ms. Interessant zu sehen ist, dass Fristverletzungen vor der Modus-Umschaltung tatsächlich stattfinden, jedoch nur solange sich das System in Überlast befindet. Dabei traten diese bei der Empfänger-Task nach und bei der Darsteller-Task vor der tatsächlichen Modus-Umschaltung auf. Aus diesem Grund wurden die Fristverletzungen bei der Empfänger-Task für den neuen Modus gezählt, während bei der Darsteller-Task nach dem Wechsel so gut wie keine Verletzungen auftreten. Dieses Szenario weist nochmal die Wirkungsweise der automatischen Adaptions-Mechanismen von ARTOS für den Umgang mit Engpässen nach, diesmal jedoch am Beispiel einer realen Anwendung.

Average PSNR	72.08
Maximum PSNR	99.00
Minimum PSNR	7.06
Average of Values < 99 dB	44.66

Tabelle 5.5: PSNR-Ergebnisse beim Engpass-Szenario mit Degradierung

Abschließend wird der Spitzen-Signal-Rausch-Abstand zwischen dem Original und dem empfangenen Video beim Client verglichen. Die Ergebnisse sind in der Tabelle 5.5 zusammengefasst. Wie man dort sieht, ist der durchschnittliche PSNR-Wert für die gesamte Video-Sequenz mit 77.08 dB schlechter als beim vorherigen Szenario ohne Degradierung der Anwendung (vgl. Kapitel 5.7.2). Dafür fällt die durchschnittliche Qualität der Frames, die nicht mit dem Ursprungs-Signal identisch waren, deutlich besser aus. Eine Diskussion und Bewertung dieser Ergebnisse findet nun im folgenden Kapitel 5.7.4 statt.

(a) Empfänger-Task

(b) Darsteller-Task

Abbildung 5.38: Task-Kosten beim Engpass-Szenario mit Degradierung

5.7.4 Diskussion der Ergebnisse

Bei näherer Betrachtung sorgt der schlechtere PSNR-Wert bei der Degradierung der VoD-Anwendung nicht für Verwunderung. Bei den Experimenten im Kapitel 5.7.2 profitiert die Anwendung davon, dass ausschließlich I-Frames übertragen wurden. Wenn ein Frame erfolgreich empfangen und decodiert werden kann, erzeugt es automatisch einen PSNR-Wert von 99 dB, da es mit dem Original identisch ist. Die Anzahl an Fristverletzungen reicht bei diesem Szenario nicht aus, um die Gesamtqualität des wiedergegebenen Videos weiter abzusenken. Außerdem sind viele engpass-bedingte Deadline-Misses innerhalb der künstlichen Test-Applikation statt in der VoD-Anwendung aufgetreten. Bei einem realen System ohne die Fähigkeiten von ARTOS könnte die Überlastung noch größer ausfallen, so dass die VoD-Anwendung überhaupt keine Bilder mehr empfangen und zeitnah darstellen kann. In diesem Fall würde sich eine Zunahme der Verschlechterung der Gesamtqualität ergeben.

Um den erzeugten Nutzen des Systems besser evaluieren zu können, muss die Qualität des dargestellten Video-Signals stationär betrachtet werden. Beim Degradierungs-Szenario im Kapitel 5.7.3 kommen die I-Frames im Modus 2 grundsätzlich mit einem höheren PSNR-Unterschied zum Original-Video beim Empfänger an. Dadurch fällt der durchschnittliche PSNR-Wert schlechter aus, obwohl die Qualität in den Abschnitten, bei denen im ersten Szenario Bild-Material verworfen wurde, insgesamt besser ist. Dies ist aus der mit 44.66 dB höheren durchschnittlichen Qualität der Frames, die nicht mit dem Ursprungs-Signal identisch waren, abzuleiten. Darüber hinaus treten beim Degradierungs-Szenario so gut wie keine Deadline-Misses auf, was seltener zum Verlust von Daten und eine kontinuierlichere Visualisierung des Video-Stroms führt. Würde man hier eine Video-Codierung mit B- und P-Frames einführen, würde die Quote von korruptem Video-Signal höchstwahrscheinlich viel geringer ausfallen als beim Szenario ohne Degradierung, was auch zu einem besseren PSNR-Wert führen würde.

Obwohl bei der Evaluation der VoD-Anwendung die durchschnittliche Qualität bei einem Engpass durch eine Degradierung der Anwendung in einer niedrigeren Qualitätsstufe nicht verbessert werden konnte, beweisen die Beispiele, dass das Verfahren eine bessere Qualität bei potentiellen Anforderungsverletzungen erzeugt. Somit fällt bei diesem Anwendungsszenario die durchschnittliche Qualität an den Stellen, bei denen ohne Degradierungs-Mechanismen Fristverletzungen auftreten, mit Modi-Umschaltungen bis zu 40 dB besser aus. Dieses Ergebnis ist selbstverständlich szenario-spezifisch und hängt von der Definition der Qualitätsmodi der Anwendungen und der aktuellen Ressourcenbelegung des Systems ab.

Im Kapitel 1.4 wurde die Frage gestellt, wie nah ein ARTOS-System bei einem realen Anwendungsszenario an die harte Echtzeit herankommt, d.h. können Fristverletzungen trotz Engpässe

völlig vermieden werden? Wie wir beim Degradierungs-Beispiel im letzten Kapitel 5.7.3 gesehen haben, konnten die Fristverletzungen innerhalb der Tasks der VoD-Anwendung trotz der Umschaltung in einem niedrigeren Modus nicht völlig vermieden werden. Diese wurden jedoch für den Gesamtablauf des Szenarios gegenüber der Ausführung ohne Degradierungs-Mechanismen deutlich verringert. Außerdem traten Fristverletzungen überhaupt auf, weil bei dieser Anwendung der Engpass so lange erhalten blieb, bis die Anwendung alle zwischengepufferten Frames ihres aktuellen Modus abgearbeitet hat. Eine Degradierung oder gar Abschaltung der zweiten Anwendung wurde durch eine Verankerung in ihrem erstem Modus verhindert.

In einem realen System mit mehreren Anwendungen werden die Modi üblicherweise nicht fest vorgegeben, sondern anhand ihres Verbrauchs dynamisch entschieden. Eine instantane Degradierung einer anderen Applikation neben der VoD-Anwendung könnte also den Engpass beseitigen, sobald dieser vom System detektiert wurde. Die Auftrittswahrscheinlichkeit von Fristverletzungen hängt dann von der Reaktivität des Systems und von der Rekonfigurationsdauer der Anwendungen ab. Für die Reaktivität des Systems spielen diverse Faktoren bei der Approximation der Task-Kosten und ihre periodische Überprüfung eine wichtige Rolle, die bereits in Kapitel 5.4 diskutiert wurden. Insbesondere können hier die Verzögerungen durch die Rekonfigurationen der Anwendungen für ein Fortbestehen eines Bottlenecks sorgen, was im Voraus nicht absehbar ist. Solange sich jedoch das System in einem stabilen Zustand befindet und die Anwendungen approximiert sind, besteht bei konstantem Schwankungsverhalten der Tasks keine Gefahr für Fristverletzungen.

Im Folgenden Kapitel 6 schließt diese Promotionsarbeit mit einer Zusammenfassung und Bewertung der erzielten Ergebnisse ab.

6

Abschluss

Im Rahmen dieser Arbeit wurde ein offenes Echtzeit-System ARTOS entwickelt, das in der Lage ist, mit einer dynamischen Menge von Anwendungen mit unbekannten und zur Laufzeit schwankenden Ressourcenverbräuchen umzugehen. Hierfür wurde im Kapitel 3.1 ein spezielles Applikationsmodell bereitgestellt, mit dem Anwendungen und ihre Aktivitäten in mehreren Betriebsmodi mit unterschiedlichen Ressourcenanforderungen und Nutzen entwickelt und ihre Echtzeitanforderungen erfasst werden können. Die Anwendungen werden zur Laufzeit durch Entscheidungen des Systems automatisch in ihren geeignetsten Modi geschaltet, wobei Applikationen auch temporär deaktiviert werden können, um Engpässe zu umgehen. ARTOS erzielt dabei stets den höchstmöglichen Nutzen der Anwendungen bei einer optimalen Auslastung der verfügbaren Ressourcen innerhalb der verwendeten Plattform. Bei der Auswahl der optimalen Konfiguration befolgt das System auch Prioritäten der Anwendungen, welche von einem Benutzer oder dem System selbst abgegeben werden können. Dadurch realisiert ARTOS ein reaktives System, welches sich automatisch an die gegebenen Last-Bedingungen innerhalb einer Plattform anpassen kann und eine Admissionskontrolle sowie eine zentral gesteuerte Degradierungs-Strategie der Anwendungen für den Umgang mit im Voraus nicht absehbaren Engpässen bereitstellt.

Als zentrale Ressource wurde in dieser Arbeit nur die von den Anwendungen angeforderte Rechenzeit in Betracht gezogen. Die entwickelten Mechanismen lassen sich jedoch für beliebige

Ressourcentypen anwenden, wie z.B. für die verfügbare Netzwerk-Bandbreite innerhalb eines Systems oder die verfügbare Energiekapazität innerhalb eines mobilen Geräts. Eine Erweiterung der unterstützten Ressourcentypen resultiert in einer Vergrößerung des im Kapitel 3.2.3 vorgestellten Optimierungsproblems, dessen Lösung die optimale Konfiguration der Anwendungen im System ermittelt. Eine solche Erweiterung und entsprechende Lösungsmechanismen wurden im Rahmen dieser Arbeit nicht adressiert, sind jedoch innerhalb zukünftiger Forschungsarbeiten geplant.

Für den Umgang mit den zur Laufzeit schwankenden Rechenzeitanforderungen der Anwendungen wurde im Kapitel 3.2 ein dreistufiges hierarchisches Scheduling-Modell entwickelt, bei dem Informationen, wie bei einer kaskadierten Regelung, von der untersten Ebene zu den höheren Ebenen weitergeleitet und abstrahiert werden. Auf der untersten Ebene des Modells werden die Anwendungs-Tasks mit EDF gesteuert und ihre zur Laufzeit gemessenen Kosten approximiert. Entsprechende Approximationsverfahren wurden im Kapitel 3.1 diskutiert und bewertet. In ARTOS wurde ein Mechanismus gewählt, welcher Schwankungen innerhalb der gemessenen Kosten berücksichtigt und entsprechende Schätzungen mit adaptiven Ressourcenpuffern verwaltet.

Aus den ermittelten Kostenschätzungen der Tasks werden dynamisch anwendungsspezifische Ressourcenkapazitäten abgeleitet, welche im System durch den Einsatz von Constant Bandwidth Servern und ein entsprechendes Applikations-Scheduling zur Laufzeit durchgesetzt werden. Einer der Hauptbeiträge dieser Arbeit ist ein Verfahren, mit dem die Kapazitäten der Server dynamisch durch eine anwendungs-lokale Planbarkeitsanalyse an die tatsächlichen Anforderungen der Tasks angepasst werden können (vgl. Kapitel 3.2.2). Dafür wurden auch die klassischen CBS-Mechanismen so angepasst, dass durch sie mehrere Tasks gleichzeitig bedient und somit die Ressourcenverbräuche ganzer Anwendungen voneinander isoliert werden können. Eine derartige Erweiterung der CBS existiert bislang noch nicht. Die entsprechende Dimensionierung eines Servers für eine anwendungs-lokale Task-Menge wurde im Kapitel 3.2.2 beschrieben.

Die Approximationen der Task-Kosten ermöglichen eine Verzögerung der Planungsaktivitäten innerhalb der höheren Ebenen des Modells, welche mit einem zusätzlichem Aufwand für das System verbunden sind. Auf dieser Basis schlägt ARTOS ein Modell vor, mit dem ein System trotz Schwankungen in einen optimalen und stabilen Zustand überführt werden kann, solange die Approximationen der Tasks gültig sind. Verletzen Tasks ihre zuvor initialisierten Ressourcengrenzen mit einer bestimmten Rate, dann entstehen neue Schätzungen ihrer Kosten und das System muss die Planbarkeit der betroffenen Anwendungen erneut analysieren. Für den Planbarkeitstest kann prinzipiell ein beliebiges Verfahren verwendet werden. Daraus ergeben sich im vorgeschlagenen Modell neue Ressourcenkapazitäten für die aktuellen Modi der Anwendungen, die eine Optimierung ihrer aktuellen Konfiguration notwendig machen. Der Aufwand für die

Planbarkeitstests wird somit unabhängig vom jeweiligen Verfahren zeitlich verteilt und räumlich für die betroffenen Anwendungen beschränkt. Auf diese Weise wird auch die Anzahl der Optimierungen der Anwendungsmodi minimiert, da diese nur bei Änderungen der Ressourcenanforderungen der Anwendungen notwendig sind. Dieses Modell wurde im Rahmen eines Beitrags [17] auf der RTAS'15-Konferenz in Seattle vorgestellt.

Wie die Evaluation im Kapitel 5.5 gezeigt hat, können mit den Schätzungen der Kosten selbst zyklische Schwankungen der Tasks mit nur wenigen Analyse- und Rekonfigurations-Schritten approximiert werden. Die vom System erkennbare Schwankungs-Periode hängt dabei von der Größe eines adaptiven Fensters für die Berechnung der Standardabweichung innerhalb der gemessenen Task-Kosten ab. Die Größe des Fensters kann dynamisch so angepasst werden, dass eine möglichst minimale Überprovisionierung der Kosten entsteht. Die entwickelten Mechanismen wurden bereits im Rahmen eines wissenschaftlichen Journals „Informatik aktuell“ veröffentlicht [15] und auf der gemeinsamen Tagung der GI-Fachausschüsse Echtzeitsysteme und Betriebssysteme in Boppard am Rhein vorgestellt. Sie wurden dort mit dem „Best Paper Award“ ausgezeichnet.

Die zur Laufzeit ermittelten Ressourcenkapazitäten für die unterschiedlichen Modi der Anwendungen werden an die höchste Ebene im Scheduling-Modell und der dort angesiedelten Optimierung der Modi weitergereicht. Wie die Ergebnisse der Evaluationen in den Kapiteln 5.4 und 5.6 gezeigt haben, ist ARTOS nicht nur in der Lage eine optimale Konfiguration der Anwendungen mit möglichst gleichmäßiger Verteilung der Ressourcen zu finden, sondern erledigt diese Aufgabe auch innerhalb einer polynomiell begrenzten Zeit, abhängig von einer zugelassenen Menge an Applikationen, ihren Modi und einer bestimmten Ressourcenauflösung im System. Die Ergebnisse wurden bereits im Rahmen eines technischen Berichts in [14] veröffentlicht.

Die vorgeschlagenen Mechanismen wurden mit der Real-Time Specification for Java (RTSJ) implementiert und innerhalb eines komponentenbasierten Java-Frameworks integriert, bei dem Anwendungen dynamisch aus einzelnen Komponenten zusammengesetzt werden. Somit wurde ein funktionsfähiger Prototyp des ARTOS-Systems bereitgestellt, mit dem die Wirksamkeit des Modells evaluiert und bewiesen werden konnte. Allerdings hat sich RTSJ nur bedingt als eine geeignete Implementierungs-Grundlage erwiesen, da dort bestimmte Funktionen für die Realisierung von benutzerspezifischen Server-Mechanismen und Task-Scheduling fehlen. Aus diesem Grund wurden spezielle Erweiterungen von RTSJ entwickelt, die in Zusammenarbeit mit der Firma Aicas innerhalb der JamaicaVM prototypisch implementiert und integriert wurden. Die korrekte Funktionsweise der modifizierten JVM und des darauf aufbauenden Scheduling-Modells von ARTOS wurde bei den Evaluations-Szenarien in den Kapiteln 5.2 und 5.3 nachgewiesen. Darüber hinaus wurden die erarbeiteten RTSJ-Mechanismen zur Standardisierung vorgeschlagen. Da zum Zeitpunkt der Bearbeitung dieser Dissertation eine neue Version der Spezifikation

erstellt wurde, konnten die vorgeschlagenen Mechanismen direkt in die neue Version aufgenommen werden. Die Diskussionen mit der RTSJ-Expert Group fanden unter anderem im Rahmen der JTRES'15-Konferenz in Paris statt, wo die vorgeschlagenen RTSJ-Erweiterungen im Kontext des hierarchischen Scheduling-Modells von ARTOS publiziert und präsentiert wurden.

Im Rahmen dieser Arbeit wurde auch eine mit dem ARTOS-System kompatible Video-on-Demand Applikation entwickelt, welche im Kapitel 5.7 für die Evaluation des durch ARTOS erlangten Qualitätsgewinns innerhalb eines realen Anwendungsszenarios herangezogen wurde. Die Ergebnisse dieser Evaluation zeigten, dass ARTOS in diesem konkreten Fall zwar nicht in der Lage war, die durchschnittliche Qualität eines übertragenen Videos durch eine Degradierung der Anwendung und Vermeidung von temporalen Fehlern zu verbessern, jedoch für die Stellen an denen Fehler auftraten eine deutlich bessere Qualität erzielt werden konnte. Der Verbesserungsfaktor ist abhängig von der tatsächlichen Ausprägung der Qualitätsunterschiede zwischen den unterschiedlichen Anwendungsmodi und von der Anzahl der aufgetretenen Fehler.

Die in Kapitel 5 betriebenen Evaluationen konnten zudem zeigen, dass 1.) das System schnell eine optimale Selektion von Modi entsprechend den Anwendungsprioritäten etabliert, 2.) es planbar bleibt, solange Tasks approximiert sind, 3.) Fristverletzungen nur dann auftreten, wenn die Tasks ihre Sicherheitspuffer verletzen, 4.) Planbarkeitstests nur dann gemacht werden, wenn Vorhersagen nicht mehr aktuell sind, 5.) die kontrollierte Degradierung das Auftreten von Fristverletzungen reduziert und 6.) zyklische Überlast von Tasks automatisch durch die Sicherheitspuffer erfasst und somit periodische Rekonfigurationen des Systems unterdrückt werden. Somit wurden alle ursprünglich im Kapitel 1.3 definierten Anforderungen an die dynamische Ressourcensteuerung durch das ARTOS-System erfüllt.

6.1 Zukünftige Arbeiten

Die Vorstellung der potentiellen zukünftigen Arbeiten gliedert sich in Erweiterungen des vorgeschlagenen Scheduling-Modells und in einer Diskussion seiner potentiellen Einsatzgebiete. Zunächst werden mögliche Verbesserungen des Modells vorgestellt, wobei hier nur einige der prominentesten Ansätze präsentiert werden. Die Arbeit endet in diesem Kapitel mit einer generellen Diskussion der möglichen Einsatzgebiete des entwickelten Scheduling-Modells und seiner Mechanismen.

Erweiterungen des Modells

Statt dem für die Approximation der Rechenzeitanforderung der Tasks vorgeschlagenen Algorithmus von Baruah [28] (siehe Kapitel 3.2.2), kann ein effizienteres Verfahren für die anwendungs-lokale Planbarkeitsanalyse verwendet werden, wie zum Beispiel der *All Approximated*-Algorithmus von Albers und Slomka [29]. Dieses Verfahren ermöglicht nicht nur einen schnelleren Nachweis der Planbarkeit einer Task-Menge, sondern auch eine präzisere Approximation ihrer kumulativen Rechenzeitanforderung. Auf diese Weise kann eine präzisere Dimensionierung der Anwendungs-CBS mit einer minimalen Überprovisionierung der tatsächlichen Task-Kosten erfolgen, was zu einer besseren Auslastung des Systems führen kann.

Für eine feingranulare Verteilung nicht genutzter Server-Kapazitäten bei vorzeitigen Beendigungen der Anwendungs-Tasks, die noch nicht durch ihre Sicherheitspuffer erfasst wurden, können die bereits im Kapitel 2.2.1 diskutierten CASH-Mechanismen [39] auf der CBS-Ebene integriert werden. Diese verteilen die nicht genutzten Kapazitäten instantan an andere Server und Anwendungen, wodurch diese weniger wahrscheinlich in Engpässe geraten. CASH sorgt also für eine Verteilung der freien Ressourcen, die durch Schwankungen der Task-Kosten innerhalb ihrer Approximationspuffer entstehen.

Darüber hinaus kann das Modell mit beliebigen anwendungsspezifischen Scheduling-Mechanismen (statt nur EDF) auf der Task-Ebene erweitert werden. Dafür müssten die Scheduling-Schnittstellen des Modells den Anwendungen entsprechend verfügbar gemacht werden. Diese Erweiterung erfordert jedoch eine Verlagerung der Planbarkeitsanalyse und der Kapazitäts-Bemessungen der Server innerhalb der Anwendungen, was zu einer schlechteren Reaktivität des Systems führen kann.

Die bei der Schätzung der Task-Kosten verwendete doppelte Standardabweichung ermöglicht eine Approximation ihrer Schwankungen anhand der beobachteten Historie mit einer 95.4 %-igen Güte (vgl. Kapitel 3.1.2). Als alternatives Verfahren könnte hier ein Quantil verwendet werden, was eine effizientere Approximation mit derselben Güte verspricht. Fallen die Puffer sehr groß aus, so kann bei der Approximation der Kosten eine andere Verteilung, als die in dieser Arbeit vorausgesetzte Normalverteilung, besser geeignet sein. Hier kann zum Beispiel mit dem Kolmogorow-Smirnow-Test überprüft werden, ob die gemessenen Kosten einer Normalverteilung folgen. Mit der Kolmogorow-Ungleichung kann eine von der Verteilung unabhängige untere Schranke ermittelt werden, wieviele der gemessenen Kosten-Samples innerhalb der n -fachen Standardabweichung liegen. Ist die Approximationsgüte des Systems verletzt, kann ein Histogramm der tatsächlichen Verteilung berechnet und eine spezifische Schätzung der Abweichung ermittelt werden.

Wie bereits erwähnt, kann das Optimierungsverfahren für die Konfiguration der Anwendungsmodi mit zusätzlichen Dimensionen, wie etwa andere Ressourcen-Typen, erweitert werden. Darüber hinaus können auch andere Entscheidungsgrundlagen mit in die Optimierung aufgenommen werden, wie zum Beispiel die optimale Platzierung einer Task innerhalb eines Multicore- bzw. Multiprozessor-Systems. Dort können die Kosten für die Migration einer Task oder einer ganzen Anwendung auf einem anderen Rechenkern bzw. Rechenknoten mit ihrer aktuellen Rechenzeit-Anforderung abgewogen werden. Ein solches Verfahren erweitert das ARTOS-System mit Fähigkeiten für ein verteiltes, qualitäts-basiertes Scheduling mit automatischen Mechanismen zur Lastbalancierung.

Mögliche Einsatzgebiete

Das entwickelte Scheduling-Modell und die Mechanismen in dieser Arbeit finden prinzipiell in allen Systemen eine Anwendung, bei denen Ressourcen dynamisch gesteuert und auch Echtzeitanforderungen von Anwendungen erfüllt werden müssen. Diese können zum Beispiel Cloud- oder HPC-Systeme innerhalb großer Datenzentren mit hunderten von vernetzten Prozessor-Einheiten sein, aber auch mobile und eingebettete Geräte mit beschränkten Ressourcen und Rechenkapazitäten. In beiden Fällen kann die verfügbare Ressourcenkapazität des Systems, unabhängig vom tatsächlichen Deployment der Anwendungen, mit ARTOS modelliert und optimal ausgelastet werden. Der Einsatz von ARTOS im Rahmen dynamisch gesteuerter Cloud-Systeme wurde bereits beim 6th Cloud Control Workshop im Rahmen der UCC 2014-Konferenz in London vorgestellt [1]. Dort wurde auch eine Spezialisierung der Optimierung der Anwendungs- und Task-Modi vorgeschlagen, mit der ein Service Level Agreement (SLA) für die Antwortzeit der bereitgestellten Dienste von einem Cloud-Provider bestmöglichst erfüllt werden kann.

Das vorgestellte hierarchische Scheduling-Modell ließe sich grundsätzlich in ein Betriebssystem integrieren. Lelli et al. haben bereits einen `SCHED_DEADLINE` genannten EDF-Scheduler [23, 24] für einen Linux-Kernel entwickelt. Dieser wurde im Rahmen des JUNIPER-Projekts [25] als Basis für einen hierarchischen Scheduler [26] für Multiprozessoren verwendet. Basierend auf `SCHED_DEADLINE` kann auch das vorgestellte Modell realisiert werden. Durch eine spezielle API im *User-Space* können Applikationen ihre Modi und Tasks im Kernel registrieren. Ein Modus-Wechsel kann als Folge eines speziellen Systemaufrufs ausgelöst werden, z.B. wenn sich Tasks am Ende eines Jobs selbst suspendieren. Eine Integration der ARTOS-Mechanismen auf der Basis von `SCHED_DEADLINE` im Linux-Kernel ist im Rahmen eines Forschungsprojekts zusammen mit der Scuola Superiore Sant'Anna und Aicas in Planung.

Darüber hinaus soll ARTOS als Basis für ein Forschungsprojekt im Rahmen von HPC-Systemen dienen und dort für eine Performanz-Steigerung durch dynamische Ressourcen-Steuerung sorgen. Bei diesem Projekt kommen auch die weiter oben diskutierten Erweiterungen der Optimierung mit Entscheidungen über die optimale Platzierung von Tasks zum Tragen. Eine Integration des Modells innerhalb des MyThOS-Betriebssystems soll eine Basis-Plattform für das geplante Forschungs-Projekt bereitstellen. Durch den minimalen Overhead von MyThOS können jedoch dann auch eingebettete Systeme als Ausführungs-Plattformen anvisiert werden, bei denen eine adaptive Planung der verfügbaren Ressourcen, unter Beachtung von Echtzeitanforderungen, besonders von Bedeutung ist. Mit der erweiterten ARTOS-Version kann die Ressourcenauslastung innerhalb mehrerer Endgeräte oder Systeme optimiert und balanciert werden, was ARTOS für den Einsatz innerhalb IoT-Szenarien vorbereitet.

Literaturverzeichnis

- [1] V. Nikolov, S. Kächele, F. J. Hauck, and D. Rautenbach. CLOUDFARM: An Elastic Cloud Platform with Flexible and Adaptive Resource Management. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, UCC '14*, pages 547–553. IEEE Computer Society, 2014.
- [2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.
- [3] W. C. Hsieh, P. Wang, and W. E. Weihl. Computation Migration: Enhancing Locality for Distributed-memory Parallel Systems. *SIGPLAN Not.*, 28(7):239–248, 1993.
- [4] F Mattern and C. Floerkemeier. From Active Data Management to Event-based Systems and More. chapter From the Internet of Computers to the Internet of Things, pages 242–259. Springer-Verlag, 2010.
- [5] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac. Internet of Things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516, 2012.
- [6] F. Travostino, P. Daspit, L. Gommans, C. Jog, C. de Laat, J. Mambretti, I. Monga, B. van Oudenaarde, S. Raghunath, and P. Y. Wang. Seamless Live Migration of Virtual Machines over the MAN/WAN. *Future Generation Computer Systems*, 22(8):901–907, 2006.
- [7] C. Isci, J.E. Hanson, I. Whalley, M. Steinder, and J.O. Kephart. Runtime Demand Estimation for effective dynamic resource management. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pages 381–388, 2010.
- [8] G. Lee, B.-G. Chun, and H. Katz. Heterogeneity-aware Resource Allocation and Scheduling in the Cloud. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud’11, pages 4–4. USENIX Association, 2011.

- [9] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. *SIGPLAN Not.*, 48(4):77–88, 2013.
- [10] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: a resource-centric approach to real-time systems. In *Proc. of the Conf. on Multimedia Comp. and Networking—MMCN*, 1998.
- [11] P. Dibble. *Real-time JAVA platform programming*. Prentice Hall, 2002.
- [12] G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency*. Plenum Publishing Co., 2005.
- [13] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-case Execution-time Problem – Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008.
- [14] V. Nikolov, M. Matousek, D. Rautenbach, L. Draque Penso, and F. J. Hauck. ARTOS: system model and optimization algorithm. Technical Report VS-R08-2012, Institute of Distributed Systems, University of Ulm, Ulm, 2012.
- [15] V. Nikolov, F. J. Hauck, and L. Schubert. Ein hierarchisches Scheduling-Modell für unbekannte Anwendungen mit schwankenden Ressourcenanforderungen. In *Betriebssysteme und Echtzeit*, Informatik aktuell, pages 49–58. Springer Berlin Heidelberg, 2015.
- [16] M. A. Rivas and M. G. Harbour. POSIX-compatible application-defined scheduling in MaRTE OS. In *Proc. of the 24th Euromicro Conf. on Real-Time Sys.—ECRTS*, page 67. IEEE Comp. Soc., 2002.
- [17] V. Nikolov, K. Kempf, F. J. Hauck, and D. Rautenbach. Distributing the Complexity of Schedulability Tests. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS 2015. IEEE Computer Society, 2015.
- [18] MyThOS - Operating System for Massively Threaded Applications.
<https://gauss-allianz.de/en/project/title/MyThOS>, March 2016.
- [19] G. C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications*. Springer, 2004.
- [20] C. Liu and J. W. Layland. Scheduling algorithms for multi-programming in a hard real-time environment. *Journal of The ACM*, 1973.

- [21] G. C. Buttazzo. Rate Monotonic vs. EDF: Judgment Day. *Real-Time Systems*, 29(1):5–26, 2005.
- [22] L. Sha, T. F. Abdelzaher, K.-E. Årzén, A. Cervin, T. P. Baker, A. Burns, G. C. Buttazzo, M. Caccamo, J. P. Lehoczky, and A. K. Mok. Real time scheduling theory: a historical perspective. *Real-Time Sys.*, 28:101–155, 2004.
- [23] J. Lelli, D. Faggioli, and T. Cucinotta. An efficient and scalable implementation of global EDF in Linux. In *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2011.
- [24] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli. Deadline scheduling in the Linux kernel. *Software: Practice and Experience*, 2015.
- [25] JUNIPER - Java Platform for High-Performance and Real-Time Large Scale Date. <http://www.juniper-project.org/>, July 2015.
- [26] G. Lipari and E. Bini. A Framework for Hierarchical Scheduling on Multiprocessors: From Application Requirements to Run-Time Allocation. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 249–258, Nov 2010.
- [27] G. Bolella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The real-time specification for Java*. Addison-Wesley, 2000.
- [28] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 182–190, Dec 1990.
- [29] K. Albers and F. Slomka. Efficient feasibility analysis for real-time systems with EDF scheduling. In *DATE, 2005. Proceedings*, 2005.
- [30] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems, 1994.
- [31] L. Abeni and G. C. Buttazzo. Integrating multimedia applications in hard real-time systems. In *IEEE Real-Time Sys. Symp.*, pages 4–13, 1998.
- [32] R.I. Davis, K.W. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *Real-Time Systems Symposium, 1993., Proceedings.*, pages 222–231, 1993.
- [33] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. In *IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

- [34] J.K. Strosnider, J.P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Comp.*, 44(1):73–91, 1995.
- [35] J.P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Real-Time Systems Symposium, 1992*, pages 110–123, 1992.
- [36] M. Spuri and G. C. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. *Real-Time Systems*, 10:179–210, 1996.
- [37] G. Lipari and S. K. Baruah. Greedy reclamation of unused bandwidth in constant-bandwidth servers. In *Proc. of the 12th Euromicro Conf. on Real-Time Sys.—ECRTS*, pages 193–200, 2000.
- [38] M. Caccamo, G. C. Buttazzo, and D. C. Thomas. Efficient reclaiming in reservation-based real-time systems with variable execution times. *IEEE Trans. on Comp.*, 54, February 2005.
- [39] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proc. of the 21st IEEE Real-Time Sys. Symp.—RTSS*, pages 295–304, 2000.
- [40] K.-E. Arzen, A. Cervin, J. Eker, and L. Sha. An introduction to control and scheduling co-design. In *Conf. on Decision and Control*, volume 5, 2000.
- [41] P. Zhou, J. Xie, and X. Deng. Optimal feedback scheduling of model predictive controllers. *J. of Control Theory and Appl.*, 4:175–180, 2006.
- [42] J. Eker, P. Hagander, and K. Årzén. A Feedback Scheduler for Real-Time Controller Tasks. *Proc. of The IEEE*, 2000.
- [43] J. A. Stankovic, C. Lu, S. H. Son, and G. Tao. The case for feedback control real-time scheduling. In *Euromicro Conf. on Real-time Sys.*, pages 11–20, 1999.
- [44] C. Lu, J. A. Stankovic, T. F. Abdelzaher, G. Tao, S. H. Son, and M. Marley. Performance specifications and metrics for adaptive real-time systems. In *Proc. of the 21st IEEE Real-Time Sys. Symp.—RTSS*, pages 13–24, 2000.
- [45] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao. Feedback control real-time scheduling: framework, modeling, and algorithms. *Real-Time Sys.*, 23:85–126, 2002.

- [46] L. Palopoli, L. Abeni, T. Cucinotta, G. Lipari, and S. K. Baruah. Weighted feedback reclaiming for multimedia applications. In *Emb. Sys. for Real-Time Multimedia*, pages 121–126, 2008.
- [47] L. Abeni and G. Buttazzo. Adaptive bandwidth reservation for multimedia computing. In *Proc. of the 6th IEEE Int. Conf. on Emb. and Real-Time Comp. Sys. and Appl.—RTCSA*, 1999.
- [48] L. Abeni, L. Palopoli, G. Lipari, and J. Walpole. Analysis of a reservation-based feedback scheduler. In *Proc. of the 23th IEEE Real-Time Sys. Symp.—RTSS*, 2002.
- [49] T. Cucinotta, L. Abeni, L. Palopoli, and G. Lipari. A Robust Mechanism for Adaptive Scheduling of Multimedia Applications. *ACM Trans. on Embed. Comput. Syst.*, 10(4):1–24, 2011.
- [50] G. Koren and D. Shasha. Skip-Over: algorithms and complexity for overloaded systems that allow skips. In *IEEE Real-Time Sys. Symp.*, pages 110–117, 1995.
- [51] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic Scheduling for Flexible Workload Management. *IEEE Trans. on Comp.*, 51:289–302, 2002.
- [52] H. Tokuda and T. Kitayama. *Dynamic QOS Control based on Real-Time Threads*. Springer, 1993.
- [53] T. Cucinotta, L. Palopoli, L. Abeni, D. Faggioli, and G. Lipari. On the integration of application level and resource level QoS control for real-time applications. *IEEE Trans. on Ind. Informatics*, 6:479–491, 2010.
- [54] J. W. S. Liu, K.-J. Lin, W. K. Shih, A. C.-S. Yu, J. Y. Chung, and W. Zhao. Algorithms for scheduling imprecise computations. *IEEE Comp.*, 24:58–68, 1991.
- [55] J. W. S. Liu, W. K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. *Proc. of The IEEE*, 82, 1994.
- [56] E. K. P. Chong and V. Zhao. Performance evaluation of scheduling algorithms for imprecise computer systems. *J. of Sys. and Softw.*, 15:261–277, 1991.
- [57] C.-J. Hamann, L. Reuther, J. Wolter, H. Härtig, J. Löser, and S. Schönberg. Quality-assuring scheduling-using stochastic behavior to improve resource utilization. In *Proc. of the 22th IEEE Real-Time Sys. Symp.—RTSS*, pages 119–128, 2001.

- [58] H. Härtig, S. Zschaler, M. Pohlack, R. Aigner, S. Göbel, C. Pohl, and S. Röttger. Enforceable component-based realtime contracts. *Real-time Sys.*, 35:1–31, 2007.
- [59] C. Lee, J. P. Lehoczky, R. Rajkumar, and D. P. Siewiorek. On quality of service optimization with discrete QoS options. In *Proc. of the 5th IEEE Real-Time and Emb. Techn. and Appl. Symp.—RTAS*, 1999.
- [60] K. Ecker, F. Drews, and J. Lichtenberg. QoS-based management of multiple shared resource in dynamic real-time systems. In *Proc. of the 20th Int. Par. and Distrib. Proc. Symp.—IPDPS*, 2006.
- [61] S. Ghosh, J. Hansen, R. Rajkumar, and J. Lehoczky. Integrated resource management and scheduling with multi-resource constraints. In *Proc. of the 25th IEEE Real-Time Sys. Symp.—RTSS*, 2004.
- [62] F. Harada, T. Ushio, and Y. Nakamoto. Adaptive resource allocation control for fair QoS management. *IEEE Trans. on Comp.*, 56:344–357, 2007.
- [63] S. A. Brandt and G. J. Nutt. Flexible Soft Real-Time Processing in Middleware. *Real-time Sys.*, 22:77–118, 2002.
- [64] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time operating systems. In *Proc. of the 6th IEEE Real-Time Sys. Symp.—RTSS*, pages 112–122, 1985.
- [65] A. Born De Oliveira and E. Camponogara. Dynamic reconfiguration in reservation-based scheduling: an optimization approach. In *Proc. of the 15th IEEE Real-Time and Emb. Techn. and Appl. Symp.—RTAS*, pages 173–182, 2009.
- [66] A. Burns, R. Davis, and A. Zabus. Schedulability analysis techniques for single processors, 2007.
- [67] M. Aldea, G. Bernat, I. Broster, A. Burns, R. Dobrin, J. M. Drake, G. Fohler, P. Gai, M. G. Harbour, G. Guidi, J. J. Gutierrez, T. Lennvall, G. Lipari, J. M. Martinez, J. L. Medina, J. C. Palencia, and M. Trimarchi. FSF: a real-time scheduling architecture framework. In *Real-Time and Emb. Techn. and Appl., IEEE Symp.*, pages 113–124, 2006.
- [68] G. Lipari, T. Cucinotta, F. Checconi, D. Faggioli, L. Abeni, L. Palopoli, and P. Valente. Design and implementation of a middleware for QoS management - II, 2005.
- [69] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. AQuoSA - adaptive quality of service architecture. *Software, Practice and Experience*, 39:1–31, 2009.

- [70] T. F. Abdelzaher, E. M. Atkins, and K. G. Shin. QoS negotiation in real-time systems and its application to automated flight control. In *Proc. of the 3rd IEEE Real-Time and Emb. Techn. and Appl. Symp.—RTAS*, pages 228–238, 1997.
- [71] W. Y. Lee, H. Kim, and H. Lee. Maximum-utility scheduling of operation modes with probabilistic task execution times under energy constraints. *IEEE Trans. on Comp.-Aided Design of Integr. Circuits and Sys.*, 28:1531–1544, 2009.
- [72] G. C. Buttazzo and L. Abeni. Adaptive workload management through elastic scheduling. *Real-Time Sys.*, 23:7–24, 2002.
- [73] S. A. Brandt, G. J. Nutt, T. Berk, and J. E. Mankovich. A dynamic quality of service middleware agent for mediating application resource usage. In *Proc. of the 19th IEEE Real-Time Sys. Symp.—RTSS*, 1998.
- [74] S. Oikawa and R. Rajkumar. Linux/RK: a portable resource kernel in Linux. In *Proc. of the 20th IEEE Real-Time Sys. Symp.—RTSS*, 1999.
- [75] V. Nikolov. A Real-Time Implementation of an AAC Encoder in DROPS. Studienarbeit, Friedrich-Alexander Universität Erlangen-Nürnberg, 2005.
- [76] ISO/IEC 13818-7:2006 - Information technology – Generic coding of moving pictures and associated audio information – Part 7: Advanced Audio Coding (AAC)), 2006.
- [77] DROPS - The Dresden Real-Time Operating System Project.
<http://os.inf.tu-dresden.de/drops/>, June 2006.
- [78] The L4 -Kernel Family. <http://os.inf.tu-dresden.de/L4/>, February 2015.
- [79] I. E. G. Richardson. *H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia*. John Wiley & Sons, Inc., 2003.
- [80] T. Wiegand, G.J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 13(7):560–576, 2003.
- [81] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol, 2002.
- [82] J. Rosenberg and H. Schulzrinne. Session Initiation Protocol (SIP): Locating SIP Servers. RFC 3263 (Proposed Standard), June 2002.

- [83] M. Handley, V. Jacobson, and C. Perkins. SDP: Session Description Protocol. RFC 4566 (Proposed Standard), 2006.
- [84] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications, 2003.
- [85] H. Schulzrinne and S. Casner. RTP Profile for Audio and Video Conferences with Minimal Control. RFC 3551 (Standard), 2003.
- [86] S. Zilberstein. Using Anytime Algorithms in Intelligent Systems. *Ai Magazine*, 17:73–83, 1996.
- [87] OSGi Alliance. OSGi service platform, Release 4.
<http://www.osgi.org/Download/Release4V42>, Sept. 2009.
- [88] N. Henze. *Stochastik für Einsteiger*. Springer Fachmedien, Wiesbaden, 10., überarb. edition, 2013.
- [89] ITU-T J.247 – Objective perceptual multimedia video quality measurement in the presence of a full reference. Technical report, 2008.
- [90] T. Zinner, O. Hohlfeld, O. Abboud, and T. Hossfeld. Impact of frame rate and resolution on objective QoE metrics. In *Quality of Multimedia Experience (QoMEX), 2010 Second International Workshop on*, pages 29–34, 2010.
- [91] J. Robert and C. Patterson. Video Encoding Settings for H.264 Excellence.
<http://www.lighterra.com/papers/videoencodingh264/>, January 2016.
- [92] FFmpeg Wiki. FFmpeg and H.264 Encoding Guide.
<https://trac.ffmpeg.org/wiki/Encode/H.264>, January 2016.
- [93] J. Brandt and L. Wolf. Impact of adaptation dimensions on video quality. In *Consumer Electronics (ISCE), 2010 IEEE 14th International Symposium on*, pages 1–6, June 2010.
- [94] S. Buchinger and H. Hlavacs. Subjective Quality of Mobile MPEG-4 Videos with Different Frame Rates. *J. Mob. Multimed.*, 1(4):327–341, 2005.
- [95] J. D. McCarthy, M. A. Sasse, and D. Miras. Sharp or Smooth?: Comparing the Effects of Quantization vs. Frame Rate for Streamed Video. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '04*, pages 535–542. ACM, 2004.

- [96] H. Schwarz, D. Marpe, and T. Wiegand. Overview of the Scalable Video Coding Extension of the H.264/AVC Standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 17(9):1103–1120, 2007.
- [97] T. F. Abdelzaher and N. T. Bhatti. Web Content Adaptation to Improve Server Overload Behavior. *Computer Networks*, 31, 1999.
- [98] J. Kim, S. Elnikety, Y. He, S. Hwang, and S. Ren. Qaco: Exploiting partial execution in web servers. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*. ACM, 2013.
- [99] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodríguez. Brownout: Building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 700–711. ACM, 2014.
- [100] V. Nikolov, F. J. Hauck, and S. Wesner. Assembling a Framework for Unknown Real-Time Applications with RTSJ. In *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '15, pages 12:1–12:10, New York, NY, USA, 2015. ACM.
- [101] J. G. Proakis and D. G. Manolakis. *Digital Signal Processing (3rd Ed.): Principles, Algorithms, and Applications*. Prentice-Hall, Inc., 1996.
- [102] Forecasting Theory - Double Exponential Smoothing.
http://www.me.utexas.edu/~jensen/ORMM/omie/operation/unit/forecast/double_exp.html, January 2016.
- [103] E. Panteley and R. Ortega. Cascaded control of feedback interconnected nonlinear systems: Application to robots with {AC} drives . *Automatica*, 33(11):1935 – 1947, 1997.
- [104] L. Abeni, G. Lipari, and J. Lelli. Constant Bandwidth Server Revisited. *SIGBED Rev.*, 11(4):19–24, 2015.
- [105] L. George, N. Rivierre, and M. Spuri. Preemptive and Non-Preemptive Real-Time Uni-Processor Scheduling. Research Report RR-2966, INRIA, 1996. Projet REFLECS.
- [106] U. Schöning. *Algorithmik*. Spektrum Akademischer Verlag, 2001.
- [107] A. Biondi, A. Melani, and M. Bertogna. Hard Constant Bandwidth Server: Comprehensive formulation and critical scenarios. In *Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on*, pages 29–37, 2014.

- [108] G. Buttazzo and E. Bini. Optimal Dimensioning of a Constant Bandwidth Server. In *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, pages 169–177, 2006.
- [109] Schmidt, H. and Elsholz, J.-P. and Nikolov, V. and Hauck, F. J. and Kapitza, R. OSGi4C: enabling OSGi for the cloud. In *Proc. of the 4th Int. ICST Conf. on Comm. Sys. Softw. and Middleware—COMSWARE*, pages 1–12. ACM, 2009.
- [110] V. Nikolov, R. Kapitza, and F. J. Hauck. Recoverable class loaders for a fast restart of Java applications. *Mobile Networks and Applications (MONET)*, 14(1):53–64, 2009.
- [111] V. Nikolov. Untersuchung von Optimierungspotentialen beim Start einer Umgebung für dynamische Softwarekomponenten am Beispiel von Java und OSGi. Master’s thesis, Universität Erlangen-Nürnberg, 2007.
- [112] Concierge OSGi. <http://concierge.sourceforge.net/>, April 2009.
- [113] T. Richardson, Andy J. Wellings, J. A. Dianes, and M. Díaz. Providing temporal isolation in the OSGi framework. In *Proc. of the 7th Int. Workshop on Java Techn. for Real-Time and Emb. Sys.—JTRES*, pages 1–10, 2009.
- [114] T. Richardson and A. Wellings. An admission control protocol for real-time OSGi. In *Proc. of the 13th IEEE Symp. on Obj.-Oriented Real-Time Distrib. Comp.—ISORC*, pages 217–224, 2010.
- [115] T. Richardson and A. J. Wellings. RT-OSGi: Integrating the OSGi Framework with the Real-Time Specification for Java. In *Distrib., Emb. and Real-time Java Sys.*, pages 293–392. Springer, 2012.
- [116] M. Joseph and P. K. Pandya. Finding response times in a real-time system. *The Comp. J.*, 29:390–395, 1986.
- [117] R. I. Davis, A. Zabos, and A. Burns. Efficient exact schedulability tests for fixed priority real-time systems. *IEEE Trans. on Comp.*, 57:1261–1276, 2008.
- [118] E. Bini and S. K. Baruah. Efficient computation of response time bounds under fixed-priority scheduling. *Real-Time Sys.*, 2007.
- [119] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proc. of the 4th ACM Int. Conf. on Emb. Software—EMSOFT*, pages 95–103, 2004.

- [120] Jamaicavm. <https://www.aicas.com/cms/en/JamaicaVM>, February 2015.
- [121] IEEE Standard for Information Technology - Portable Operating System Interfaces (POSIX(R)) - Part 1: System Application Program Interface (API) - Amendment 1: Realtime Extension [C language]. *IEEE Std 1003.1b-1993*, 1994.
- [122] Robert Love. *Linux Kernel Development*. Addison-Wesley, 2010.
- [123] aicas GmbH. *JamaicaVM 6.3 — User Manual*. aicas.
- [124] PREEMPT_RT. <https://rt.wiki.kernel.org>, January 2016.
- [125] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *Computers, IEEE Transactions on*, 39(9):1175–1185, Sep 1990.
- [126] ARTOS. <http://www.uni-ulm.de/in/vs/proj/artos.html>, February 2014.
- [127] ARTOS GitHub-Repository. <https://github.com/vladnikolov/arema>, April 2016.
- [128] A. J. Wellings and M. S. Kim. Processing Group Parameters in the Real-time Specification for Java. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '08. ACM, 2008.
- [129] A. Burns and Andy J. Wellings. Processing Group Parameters in the Real-Time Specification for Java. In *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, volume 2889 of *Lecture Notes in Computer Science*. Springer, 2003.
- [130] A. Zerzelidis and A. J. Wellings. Getting More Flexible Scheduling in the RTSJ. In *Object-Oriented Real-Time Distributed Computing*, pages 3–10, 2006.
- [131] A. Zerzelidis and A. J. Wellings. A Framework for Flexible Scheduling in the RTSJ. *ACM Trans. Embed. Comput. Syst.*, 10(1), 2010.
- [132] Realtime and Embedded Specification for Java (Verison 2.0, Draft 21, Zeppelin Edition), August 2015.
- [133] Guidelines for Extending the RTP Control Protocol (RTCP). <https://tools.ietf.org/html/rfc5968>, April 2016.
- [134] Simple DirectMedia Layer. <https://www.libsdl.org/>, March 2016.

Abbildungsverzeichnis

1.1	Beispielszenario	8
1.2	Dynamische Applikationsmenge	9
1.3	Komponentenbasierte Applikationen	9
1.4	Dienstbasierte Applikationen	10
1.5	Integration des Ressourcenmanagement-Frameworks	13
1.6	Thematische Einordnung der Arbeit	13
2.1	Task-Modell	18
2.2	Harte und weiche Echtzeit	19
2.3	Aperiodische und periodische Task-Modelle	20
2.4	Ablaufplanung mit EDF	22
2.5	Aktivierung einer Task $\tau = (T, C, D) = (3, 1, 2)$	24
2.6	Graphische Interpretation der Rechenzeitanforderungsfunktion (DBF)	25
2.7	Approximation der DBF	27
2.8	Monotone Beziehung zwischen Ressourcenverbrauch und Nutzen	48
2.9	Realisierung von Task-Modi (U = Qualität, C = Kosten)	50
3.1	Applikationsmodell	63
3.2	Beispiel eines Applikationsmodus $M_{i,j}$	65
3.3	Null-Modus einer Applikation	66
3.4	Messung der reinen Task-Kosten c_t	67
3.5	Konfidenzintervalle der Standardabweichung	73
3.6	Ebenen des hierarchischen Scheduling-Modells von ARTOS	79
3.7	Approximierter stabiler Zustand	80
3.8	Zustandsaktualisierung	81
3.9	Sicherheitspuffer mit $2 \cdot \sigma$	84
3.10	Aktualisierung eines Sicherheitspuffers (aus [15])	86
3.11	Kapselung von Anwendungen in Constant Bandwidth Servern (aus [15])	87

3.12 Funktionsweise eines CBS (Beispiel entnommen aus [31])	91
3.13 Beispiel von zwei Tasks mit einem CBS	94
3.14 Anwendungen mit Server-Perioden gleich den Task-Hyperperioden	98
3.15 Anwendungen mit Server-Perioden gleich den kleinsten Task-Perioden	100
3.16 Beispiel für $c_D(H)$ für die Task-Menge aus Abbildung 2.6	103
3.17 Graphische Interpretation der Server-Dimensionierung	104
3.18 Beispiel für ein Optimierungsszenario	110
3.19 Überblick über das Scheduling-Modell	114
4.1 OSGi-Architekturmodell [87]	118
4.2 Integration von ARTOS innerhalb von OSGi	132
4.3 Prioritätsraum für applikations-lokales Scheduling	142
4.4 EDF-Aktivierung mit Prioritätsverschiebungen (aus [100])	144
4.5 Aufteilung des Prioritätsraums von ARTOS	147
4.6 Perioden-Verkürzung mit JamaicaVM (aus [100])	170
5.1 Visualisierung des Zustands eines ARTOS-Systems	180
5.2 Steuerung eines ARTOS-System mit einer Kommando-Shell	182
5.3 Plot der Kosten einer Task	183
5.4 Jamaica-Trace	184
5.5 Beispiel einer periodisch burstenden Anwendung in ARTOS (aus [15])	186
5.6 Generische Test-Anwendung	186
5.7 Theoretischer Ablaufplan des Task-Beispiels aus Kapitel 2.1.1	189
5.8 Tatsächlicher Aktivierungsablauf des Task-Beispiels aus Kapitel 2.1.1 in ARTOS	190
5.9 Konfiguration des Evaluations-Szenarios auf der Task-Ebene	191
5.10 Systemzustand beim Experiment	191
5.11 Konfiguration des Evaluations-Szenarios auf der Applikations-Ebene	192
5.12 Kapazitätsberechnung der Applikations-Server	193
5.13 Systemzustand beim Experiment auf der Applikations-Ebene	194
5.14 Aktivierungsreihenfolge der Tasks	194
5.15 Konfiguration der Anwendungen mit mehreren Tasks	196
5.16 Kapazitätsberechnung der Applikations-Server	197
5.17 Systemzustand	197
5.18 Theoretischer Ablauf des Test-Szenarios	198
5.19 Tatsächlicher Ablauf	199
5.20 Konfiguration der künstlichen Anwendungen	201
5.21 Systemzustand nach 15 Sekunden	203

5.22	Systemzustand nach 50 Sekunden	203
5.23	Kosten-Verlauf von App 1 , Task1	204
5.24	Periodisch burstende Anwendung in ARTOS (aus [15])	207
5.25	Variation der Werte von R (aus [14])	211
5.26	Variation der Menge an Applikationen (aus [14])	213
5.27	Variation der Menge an Modi pro Applikation (1) (aus [14])	214
5.28	Variation der Menge an Modi pro Applikation (2) (aus [14])	215
5.29	Variation der Menge an Applikationen und der Ressourcengrenze (1)	215
5.30	Variation der Menge an Applikationen und der Ressourcengrenze (2) (aus [14])	216
5.31	Architeltur der VoD-Applikation	219
5.32	Modi der VoD-Applikation	220
5.33	Laufzeitkosten der Modi der VoD-Anwendung	222
5.34	Definition der künstlichen Anwendung für den Engpass	223
5.35	Systemzustand beim Engpass ohne Degradierung	223
5.36	Task-Kosten beim Engpass-Szenario ohne Degradierung	225
5.37	Systemzustand beim Engpass mit Degradierung	226
5.38	Task-Kosten beim Engpass-Szenario mit Degradierung	228

Listings

4.1	Das <code>Interruptible</code> Interface	123
4.2	Applikation	133
4.3	Applikationsmodus	133
4.4	Task-Modus	135
4.5	Periodische Task	136
4.6	Measurement-Objekt	138
4.7	Logische Prioritäten auf der Task-Ebene	141
4.8	Task-Scheduler	143
4.9	Constant Bandwidth Server-Implementierung (1)	145
4.10	Realisierung der Prioritäts-Bänder	148
4.11	Erweiterung der Task-Definition (1)	149
4.12	Implementierung des <code>CBSSchedulers</code>	150
4.13	Constant Bandwidth Server-Implementierung (2)	152
4.14	Erweiterung der Task-Definition (2)	154
4.15	Erweiterung der Task-Definition (3)	156
4.16	Constant Bandwidth Server-Implementierung (3)	157
4.17	Optimierung	159
4.18	Code-Änderungen für einen Applikationsmodus-Wechsel	161
4.19	Erweiterung der Task-Modus-Implementierung	164
4.20	Constant Bandwidth Server-Implementierung (4)	166
5.1	Implementierung eines generischen Task-Modus	188

Teile dieser Arbeit wurden publiziert in:

1. V. Nikolov and R. Kapitza. Recoverable class loaders for a fast restart of Java applications. In *Proc. of the 1st Int. Conf. on Mobile Wireless Middleware, Oper. Sys., and Appl.—MOBILWARE*. ICST, 2008.
2. V. Nikolov, R. Kapitza, and F. J. Hauck. Recoverable class loaders for a fast restart of Java applications. *Mobile Networks and Applications (MONET)*, 14(1), 2009.
3. Schmidt, H. and Elsholz, J.-P. and Nikolov, V. and Hauck, F. J. and Kapitza, R. OSGi4C: enabling OSGi for the cloud. In *Proc. of the 4th Int. ICST Conf. on Comm. Sys. Softw. and Middleware—COMSWARE*. ACM, 2009.
4. V. Nikolov, M. Matousek, D. Rautenbach, L. Draque Penso, and F. J. Hauck. ARTOS: system model and optimization algorithm. Technical Report VS-R08-2012, Institute of Distributed Systems, University of Ulm, Ulm, 2012.
5. V. Nikolov, S. Kächele, F. J. Hauck, and D. Rautenbach. CLOUDFARM: An Elastic Cloud Platform with Flexible and Adaptive Resource Management. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, UCC '14*. IEEE Computer Society, 2014.
6. V. Nikolov, K. Kempf, F. J. Hauck, and D. Rautenbach. Distributing the Complexity of Schedulability Tests. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2015*. IEEE Computer Society, 2015.
7. V. Nikolov, F. J. Hauck, and L. Schubert. Ein hierarchisches Scheduling-Modell für unbekannte Anwendungen mit schwankenden Ressourcenanforderungen. In *Betriebssysteme und Echtzeit*, Informatik aktuell. Springer Berlin Heidelberg, 2015.
8. V. Nikolov, F. J. Hauck, and S. Wesner. Assembling a Framework for Unknown Real-Time Applications with RTSJ. In *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES '15*. ACM, 2015.