CARL VON OSSIETZKY universität OLDENBURG
FAKULTÄT II
DEPARTMENT FÜR INFORMATIK

Diplomarbeit

Realtime Calculus mit hierarchischen Ereignisströmen

von

Bertram Wortelen
Matrikelnr.: 8150200

Betreut von:
Prof. Dr.-Ing. Frank Slomka (Erstgutachter)
Dipl.-Inf. Frank Bodmann (Zweitgutacher)

19. November 2007

Erklärung zur selbstständigen Arbeit

Hiermit versichere ich, die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der von mir angegebenen Quellen angefertigt zu haben. Alle aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche gekennzeichnet. Die Arbeit wurde noch keiner Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Oldenburg, den 19. November 2007

Inhaltsverzeichnis

Kapitel 1

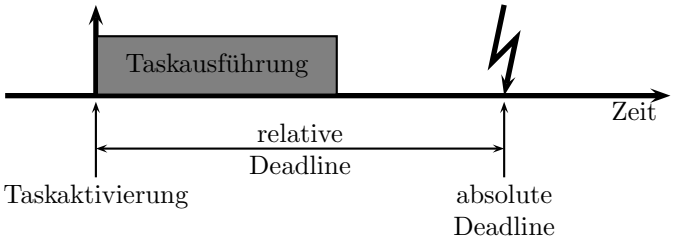
Einführung

1.1 Einleitung

In den letzten Jahren ist der Bedarf an Mikroprozessoren ständig gestiegen. Wir begegnen ihnen bewusst oder unbewusst in vielen Bereichen unseres Alltags. Sie verstecken sich dabei in Haushalts- und Multimediageräten, Fahrzeugen, Maschinen, Spielzeug und vielen mehr.
Die Aufgaben, die sie erfüllen, werden dabei immer komplexer. Damit einhergehend steigt auch die Komplexität der Prozessorsysteme, die häufig aus verschiedenen, physikalischen Teilkomponenten aufgebaut sind und über verschiedene Protokolle miteinander kommunizieren. Dabei wird die Gesamtaufgabe in mehrere Teilaufgaben (Tasks) unterteilt, die wiederum auf die Komponenten des Systems verteilt werden.
Nun hängt die Qualität eines solchen Systems nicht nur von der Güte der Algorithmen ab, die für verschiedenen Teilaufgaben zuständig sind. Es müssen auch einige zeitliche Anforderungen eingehalten werden. So ist es zum Beispiel für eine Digitalkamera wichtig, dass bei der Aufnahme eines Videos die Kamera die einzelnen Bilder schnell genug verarbeitet, damit eine hohe Bildrate erzielt werden kann. Ist dies nicht gewährleistet, so wirken Bewegungen im Film nicht flüssig.
Aus solchen Anforderungen lassen sich nun für die verschiedenen Teilaufgaben zeitliche Anforderungen ableiten, die eingehalten werden müssen. Es ist daher notwendig, den zeitlichen Ablauf (Schedule) des Systems dahingehend zu analysieren.
Für die Untersuchung des Schedules eines einzelnen Prozessors, gibt es bereits viele effiziente Algorithmen. Bei interagierenden Mehrprozessorsystemen ist es oftmals allerdings nicht ausreichend die Analyse für jeden Prozessor einzeln durchzuführen, da sich durch die Interaktion die Ablaufpläne der verschiedenen Prozessoren beeinflussen.
Daher sind hier Konzepte gefragt, die eben diese Problematik geeignet behandeln können. Insbesondere ist eine möglichst einfache Methode gesucht, das System als Ganzes zu analysieren. Eine solche Aufgabe stellt viele Herausforderungen. Vor allem muss die teilweise sehr lange Laufzeit der Analyse gehandhabt werden, da der Aufwand einer solchen Analyse sehr stark mit der Komplexität des Systems steigt.

1.2 Einordnung der Thematik

Typischerweise ist ein Mikroprozessor nicht nur zur Bearbeitung einer einzelnen Aufgabe (Task) vorgesehen, sondern hat verschiedene Aufgaben zu lösen, die sich wiederum in Teilaufgaben aufteilen können. Bei den Tasks unterscheidet man zwischen periodischen Tasks, die in regelmäßigen Zeitabständen ausgeführt werden, und nicht periodischen Tasks, die in unregelmäßigen Abständen durch äußere Ereignisse aktiviert werden. Bei den nicht periodischen Tasks wird häufig zwischen aperiodischen und sporadischen Tasks differenziert. An das zeitliche Verhalten von sporadischen Tasks werden dabei härtere Bedingungen gestellt als an aperiodische Tasks [12]. Diese Unterscheidung ist für diese Arbeit jedoch nicht von Bedeutung. Daher können die Begriffe synonym verwendet werden.
Periodische Tasks sind typischerweise in Reglersystemen oder Anwendungen mit kontinuierlichen Datenströmen, wie Audio- und Videoverarbeitung oder Netzwerkkomponenten zu finden. Sporadische Tasks sind dagegen meist Bestandteil interaktiver Anwendungen und werden zum Beispiel durch Benutzereingaben oder besondere Sensorwerte aktiviert.
Da die Tasks jedoch nicht alle zeitgleich ausgeführt werden können, muss ein Ablaufplan (Schedule) erstellt werden. Dies kann statisch während der Implementierung des Systems oder dynamisch zur Laufzeit geschehen. Bei der dynamischen Variante existiert ein weiterer Serviceprozess (Scheduler), der die Prozessorzeit auf die beteiligten Tasks verteilt. [12]
In jedem Fall muss jedoch die Bearbeitung eines Tasks vor dem Erreichen einer Zeitschranke (Deadline) abgeschlossen sein. Wird diese Zeitschranke ab dem Zeitpunkt der Taskaktivierung gezählt, so spricht man von der relativen Deadline. Die relative Deadline ist eine statische Eigenschaft eines Tasks. Ihre Größe ist abhängig von der jeweiligen Anwendung. Mit der relativen Deadline kann für jede Taskaktivierung der genaue Zeitpunkt bestimmt werden, an dem die betroffene Taskinstanz ihre Ausführung beendet haben muss. Dieser Zeitpunkt ist die absolute Deadline, die sich bei jeder Taskinstanz unterscheidet.



Für die Ablaufpläne muss also gezeigt werden, dass die Deadlines aller Tasks eingehalten werden. Für statisch generierte Pläne ist dies relativ einfach, da der Ablauf ja bereits bekannt ist. Bei dynamisch erstellten Plänen muss dagegen sichergestellt werden, dass bei jedem möglichen Ablauf der Tasks alle Deadlines eingehalten werden. Weiterhin unterscheidet man bei der Erstellung der Ablaufpläne zwischen Schedulern, die Unterbrechungen erlauben (*preemptive*) und welchen, die dies nicht tun (*non-preemptive*). Für diese Arbeit werden ausschließlich preemptive Scheduler betrachtet. [8]

Beim dynamischen Scheduling müssen die Tasks in irgendeiner Weise eine Priorität zugeteilt bekommen, anhand derer der Scheduler entscheiden kann, welcher

Task aktuell den Prozessor nutzen darf. Dabei unterscheidet man nun Varianten von dynamischer oder statischer Priorisierung. Bei Verwendung statischer Priorisierung wird die Priorität der Tasks während der Entwicklung anhand unveränderlicher Merkmale festgelegt. Sehr häufig wird die Periodizität des Tasks als ein solches Merkmal verwendet. Man spricht dann von einem Rate Monotonic Scheduler (RM). Eine weitere Alternative ist der Deadline Monotonic Scheduler (DM), der die Prioritäten statisch anhand der relativen Deadline vornimmt.
Der Earliest Deadline First Scheduler (EDF) ist dagegen ein Beispiel eines Schedulers, der eine dynamische Priorisierung der Tasks verwendet. Hier hat immer der Task die höchste Priorität, der die kleinste absolute Deadline hat.
In [14] untersuchten *Liu* und *Layland* grundlegende Eigenschaften von RM und EDF. So wiesen sie beispielsweise die Optimalität von EDF nach und ermittelten die maximalen Auslastungen, die garantiert mit dem jeweiligen Scheduler erreicht werden können. Außerdem gaben sie ein exaktes Kriterium für die Planbarkeit eines Tasksets unter EDF an. Diese Ergebnisse wurden jedoch nur für sehr einfache Tasksysteme erzielt, die aus rein periodischen Tasks bestehen. Dabei weisen diese untereinander keinerlei Abhängigkeiten auf. Zudem sind ihre Deadlines identisch mit ihrer Periodenlänge.
Diese Einschränkungen reduzieren jedoch die Menge der analysierbaren Systeme erheblich, bzw. erfordern ein hohes Maß an Abstraktion. In zahlreichen weiteren Arbeiten wurden daher Methoden entwickelt, um diese und weitere Einschränkungen zu lockern. Dabei entstanden eine Reihe von Analysemethoden für verschiedene Scheduling-Strategien und Anwendungsgebiete.
In diesem Forschungsbereich kamen nun auch Ideen zur effizienten Beschreibung von Taskaktivierungen und zur anspruchsvollen Aufgabe der Analyse von heterogenen Mehrprozessorsystemen auf. Diese Arbeit wird sich mit der Untersuchung von Algorithmen beschäftigen, die kürzlich von *Albers*, *Bodmann* und *Slomka* in [2] entwickelt wurden.

1.3 Problemstellung und Aufbau der Arbeit

Mit der Problematik, ein gesamtes Mehrprozessorsystem zu analysieren, setzt sich der Realtime Calculus auseinander. Er stellt dabei die mathematischen Grundlagen zur Verfügung, mit denen das zeitliche Verhalten für ein System untersucht werden kann, das auf das Notwendigste abstrahiert wurde.
Für diese Analyse benötigt der Realtime Calculus eine geeignete Beschreibung über das zeitliche Verhalten der Eingaben in das System. Die Beschreibung muss dabei aus Funktionen bestehen, die die Häufigkeit von Taskaktivierungen durch eine obere und eine untere Schranke abgrenzen.
Die Anforderungen, die der Realtime Calculus an diese Funktionen stellt, werden durch das Hierarchische Ereignisstrommodell erfüllt. Es wird für die Scheduling Analyse verwendet und erlaubt es Taskaktivierungen effizient und flexibel zu beschreiben.
In [2] haben sich *Albers et al.* damit beschäftigt, diese beiden Konzepte zu verknüpfen, und Algorithmen entwickelt, die die mathematischen Ideen des Realtime Calculus auf das Ereignisstrommodell anwenden.
In dieser Arbeit soll nun eine konkrete Implementierung dieser Algorithmen vorgenommen und ihre Komplexität bestimmt werden.

In Kapitel 2 wird das Konzept der Ereignisströme erläutert. Zudem wird ein spezielles Modell der Ereignisströme aufgezeigt, das in der Implementierung verwendet wurde und eine effiziente hierarchischen Darstellung der Ereignisströme erlaubt. Anschließend werden in Kapitel 3 die Ideen und Konzepte des Realtime Calculus beschrieben.
Aufbauend auf den beiden vorherigen Kapiteln soll dann in Kapitel 4 gezeigt werden, wie sich die beiden Konzepte kombinieren lassen. Hier wird auch die Vorgehensweise der in [2] vorgestellten Algorithmen erläutert.
Die konkrete Darstellung anhand der durchgeführten Implementierung der Algorithmen erfolgt jedoch erst in Kapitel 5. Dort wird auch die eigentliche Komplexitätsanalyse anhand der Algorithmen beschrieben. Dazu werden zuerst allgemeine theoretische Grundlagen zur Komplexitätsanalyse aufgezeigt. Anschließend wird dann Schritt für Schritt die Komplexität der verschiedenen Algorithmen bestimmt.

Kapitel 2

Hierarchische Ereignisströme

Im vorherigen Abschnitt wurden einige Ergebnisse vorgestellt, die bisher im Bereich der Echtzeitanalyse erzielt wurden. Viele dieser Ergebnisse setzen eine relativ beschränkte Beschreibung von Taskaktivierungen voraus. Die Ergebnisse von *Liu* und *Layland* in [14] beruhen beispielsweise auf ein Taskmodell mit ausschließlich periodischen Tasks. Es gibt zwar weitere Arbeiten, die versuchen eben diese Beschränkung zu beseitigen; allerdings erfordern auch diese Modelle zum Teil erhebliche Einschränkungen, da sie versuchen, die sporadische Taskaktivierungen in das periodische Taskmodell zu zwängen.
Um sporadische und periodische Taskaktivierungen auf natürlicherer Weise zu kombinieren, stellte *Gresser* in [12] das Ereignisstrommodell vor. Dabei werden die Taskaktivierungen über eine Liste von sporadischen oder sich periodisch wiederholenden Ereignissen beschrieben. In [1] wurde von *Albers*, *Bodmann* und *Slomka* das hierarchische Ereignisstrommodell eingeführt, das das Modell von *Gresser* syntaktisch erweitert. Mit diesem Modell ist nun eine kompaktere und flexiblere Beschreibung von Ereignisströmen möglich. In einem weiteren Schritt erweiterten sie schließlich in [2] dieses Modell, das es nun auch erlaubt, neben diskreten Ereignismustern lineare Approximationen dieser Muster anzugeben. Im Folgenden wird dieses Modell beschrieben.

2.1 Ereignisdichte

Es gibt verschiedene Wege zu prüfen, ob ein System alle zeitlichen Anforderungen einhält. Häufig werden dazu verschieden Situationen simuliert, von denen angenommen wird, dass sie das System oder einzelne Komponenten besonders auslasten. Problematisch bei der Simulation ist jedoch, dass keine Garantie dafür gegeben werden kann, dass es nicht doch weitere Situationen gibt, die nicht simuliert wurden, die aber besonders kritisch sind und die geforderten Bedingungen verletzen.[15]
Gerade im Bereich von sicherheitskritischen Systemen ist es daher notwendig, die Einhaltung kritischer Zeitanforderungen zu garantieren. Für die Echtzeitanalyse gibt es dazu verschiedene Methoden, die dies formal verifizieren. Dabei werden auf Grundlage der Systembeschreibung die kritischsten Situationen ermittelt.

Bei dieser Vorgehensweise ist die genaue zeitliche Abfolge von Ereignissen meistens gar nicht so interessant. Es ist viel eher eine geeignete Beschreibung aller Extremsituationen gefragt. Wenn durch die Analyse gezeigt werden kann, dass sich das System in den Extremsituationen korrekt verhält, so kann daraus geschlossen werden, dass dies immer so ist.
Die Ereignisströme sind nun ein Konzept, um eine solche Beschreibung zu ermöglichen. Extremsituationen werden im Normalfall durch eine hohe Anzahl an Ereignissen ausgelöst. Daher ist es nicht die Aufgabe eines Ereignisstroms, die genaue zeitliche Abfolge der Ereignisse zu beschreiben, sondern ihre zeitliche Dichte. Der Unterschied soll hier anhand eines kleinen Beispiels erläutert werden.
In Abbildung 2.1a ist das Auftreten von Ereignissen über die Zeit zu sehen. Darunter ist in Abbildung 2.1b der Ereignisstrom zu sehen, der aus dieser Ereignissequenz abgeleitet werden kann. Beim Ereignisstrom entspricht die X-Achse jedoch einer Zeitintervalllänge. Er wird gebildet, indem für das i-te Ereignis des Ereignisstroms das kleinste Intervall der Ereignissequenz gesucht wird, in dem i Ereignisse auftreten können. Das i-te Ereignis wird im Ereignisstrom nun bei der Länge dieses Intervalls eingetragen. Um das zu verdeutlichen, sind in Abbildung 2.1a die kleinsten Intervalle für 2, 3 und 4 Ereignisse eingetragen.

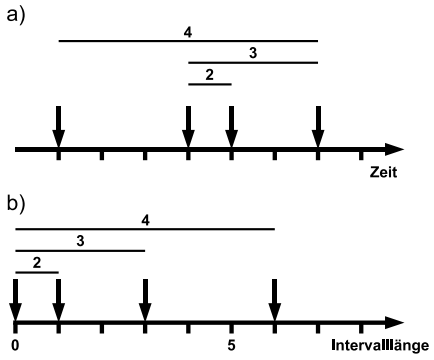


Abbildung 2.1: Ereignissequenz (a) und zugehöriger Ereignisstrom (b)

Betrachtet man nun den Ereignisstrom wieder als zeitliche Abfolge von Ereignissen, so findet man für eine beliebige Intervalllänge das Intervall mit den meisten Ereignissen immer zu Beginn des Ereignisstroms. Somit beschreibt der Ereignisstrom die maximale Ereignisdichte für beliebige Zeitintervalle.
Formal wird diese Eigenschaft in [1] mithilfe der Ereignissequenz-Funktion $ESF(I, \Theta)$ ausgedrückt. Diese Funktion liefert die Anzahl von Ereignissen der Ereignissequenz Θ innerhalb des Intervalls $[0, I]$.
Demnach ist eine Ereignissequenz Θ genau dann ein Ereignisstrom, wenn für alle Intervalle I und J gilt:

$$ESF(I + J, \Theta) \leq ESF(I, \Theta) + ESF(J, \Theta)$$

Diese Gleichung entspricht der Definition einer subadditiven Funktion. Die Subadditivität ist eine essentielle Eigenschaft der Ereignisströme, die einem bei der Arbeit mit ihnen immer bewusst sein sollte.
Eng verwandt mit der Ereignissequenz-Funktion ist die Ereignisschranken-Funktion $EBF(I, \Theta)$ - (*engl.: event bound function*). Sie liefert die maximale Anzahl an Ereignissen über ein beliebiges Intervall der Länge I der Ereignissequenz Θ. Es ist leicht ersichtlich, dass die Ereignisschranken-Funktion identisch mit der Ereignissequenz-Funktion ist, wenn Θ ein Ereignisstrom ist [1].
Die Ereignisschranken-Funktion lässt sich einfach in einem Koordinatensystem auftragen. Eine solche Darstellung ist meistens verständlicher als das Auftragen der Ereignisse auf einer Zeitachse. In Abbildung 2.2 ist das Beispiel aus

Abbildung 2.1 erneut aufgegriffen und die Ereignisschranken-Funktion des Ereignisstroms abgebildet.
In dieser Arbeit werden die Ereignisströme sehr häufig grafisch über ihre Ereignisschranken-Funktion dargestellt. Die Skalierung der X-Achse erfolgt dabei immer ohne Angabe einer Einheit, da es sich hier ausschließlich um Beispiele handelt, bei denen es uninteressant ist, ob die Werte nun in Millisekunden oder sogar Sekunden gemessen werden.

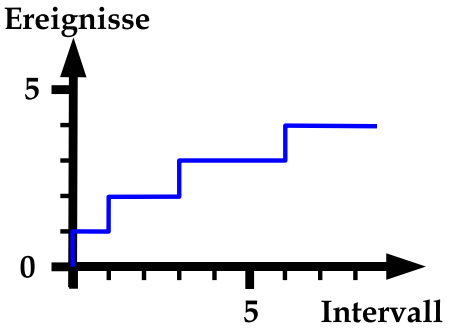


Abbildung 2.2: Beispiel einer Ereignisschranken-Funktion

2.2 Modell

Im Folgenden wird das Modell der hierarchischen Ereignisströme erläutert. Es entspricht dem, das in [2] vorgestellt wurde.
In diesem Modell wird ein Ereignisstrom Θ über eine hierarchische Baumstruktur beschrieben. Die Wurzel dieser Struktur ist der Ereignisstrom selber, dessen Kinder sich aus einer Menge von hierarchischen Ereignisstromelementen zusammensetzen:

$$\theta = (T, a, n, S)$$

Jedes Element erzeugt ein Ereignismuster. Dabei ist T die Periode, mit der sich dieses Muster wiederholt. Es lassen sich an dieser Stelle auch sporadische Ereignismuster definieren. Dies wird erreicht, indem man $T = \infty$ setzt. Hier zeigt sich jetzt auch ein Aspekt der Flexibilität dieses Models. Im Vergleich zu den Analyseverfahren, die auf ein periodisches Taskmodell aufbauen, lassen sich hier nun sporadische und periodische Taskaktivierungen durch die gleiche Struktur beschreiben.
Der zweite Paramter a ist der Offset, nach dem das erste Ereignis auftritt.
Mit n wird die Anzahl an Ereignissen, die innerhalb einer Periode durch dieses Element beschrieben werden können, beschränkt auf maximal n Ereignisse. Das eigentliche Ereignismuster wird über S beschrieben. S kann dabei entweder ein weiterer Ereignisstrom Θ' sein, der eine Hierarchieebene tiefer liegt, oder aber ein Gradient G. Elemente mit Gradienten bilden die Blätter der Struktur und beschreiben eine kontinuierlich steigende Menge von Ereignissen. Dies dient zwei Zwecken. Zum Einen wird es benötigt, um Ereignismuster approximiert zu beschreiben, und zum anderen, um einzelne, diskrete Ereignisse darzustellen. Das

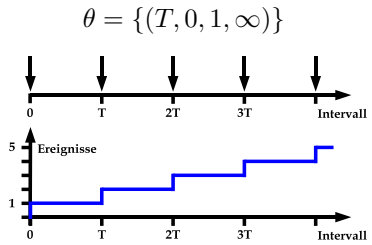


Abbildung 2.3: Beschreibung einer periodischen Taskaktivierung

lässt sich mit Elementen folgender Art bewerkstelligen:

$$\theta = (T, a, 1, \infty)$$

Durch den unendlichen Gradienten erzeugt dieses Element an der gewünschten Stelle eine unendliche Anzahl an Ereignissen. Da dieses Element jedoch eine Beschränkung von eins hat, lässt es nur die Generierung eines einzelnen Ereignisses zu. In Abbildung 2.3 ist zu sehen, wie mithilfe eines solchen Elements die periodische Aktivierung eines Tasks mit der Periode T beschrieben werden kann. Im unteren Teil der Abbildung ist der Graph der Ereignisschranken-Funktion von Θ zu sehen.
Wie bereits erwähnt, können mit dem Gradienten auch Approximationen beschrieben werden. Die Idee hierbei ist, dass man einen Ereignisstrom anfänglich exakt darstellt und nach einer gewissen Länge nur noch linear approximiert beschreibt. Nimmt man beispielsweise den Ereignisstrom $\Theta = \{(T, 0, 1, \infty)\}$ aus Abbildung 2.3, so sähe eine Approximation nach drei Ereignissen wie folgt aus:

$$\Theta_3 = \{(\infty, 0, 3, \underbrace{\{(T, 0, 1, \infty)\}}_{\Theta}), (\infty, 3, \infty, \frac{1}{T})\}$$

Das Vorgehen ist dabei sehr simpel. Der ursprüngliche Ereignisstrom Θ wird in ein sporadisches Ereignisstromelement eingebettet, das durch seine Beschränkung dafür sorgt, dass nur drei periodische Ereignisse erzeugt werden. Zusätzlich erhält Θ_3 ein weiteres Element, das den Ereignisstrom nach der periodischen Beschränkung linear approximiert mit einer Steigung von $\frac{1}{T}$ fortsetzt. Da dieses Element keine Beschränkung ($n = \infty$) hat, wird diese Gerade unendlich fortgesetzt. Das Ergebnis ist in Abbildung 2.4 zu sehen.
Wichtig dabei ist, dass es sich um eine Überapproximation handelt, da die Ereignisströme die kritischste Anzahl an Tasks für ein Zeitintervall angeben sollen.

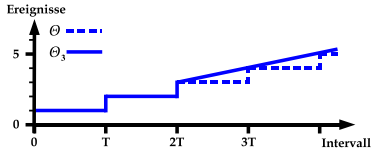


Abbildung 2.4: Approximation eines periodischen Tasks

Würde es sich nicht um eine Überapproximation handeln, so gäbe es Zeitintervalle, für die der approximierte Strom weniger Ereignisse angibt, als real auftreten können. Das würde das entsprechende Intervall weniger kritisch machen. Dies könnte wiederum dazu führen, dass die Analyse das System als planbar erkennt, obwohl es das nicht ist.
Natürlich besteht jetzt die umgekehrte Möglichkeit, dass das System fälschlicherweise als nicht planbar erkannt wird. Dieser Fall ist jedoch weniger kritisch, da es das Ziel der Analyse ist, einen Beweis zu erhalten, dass das System unter allen Umständen planbar ist. Somit erhält man durch die Überapproximation eine pessimistische Sicht des Systems.[9]
Es mag sich nun die Frage nach dem Sinn einer solchen Approximation stellen, da die Beschreibung des ursprünglichen Ereignisstroms exakt und kompakter ist. Hilfreich wird es allerdings, wenn es darum geht, mithilfe der Ereignisströme eine Planbarkeitsanalyse durchzuführen. Die Komplexität einer vollständigen und exakten Analyse steigt sehr stark mit der Komplexität des Systems. Insbesondere das Vorhandensein von periodischen Tasks führt häufig dazu, dass eine sehr große Anzahl an Instanzen dieser Tasks berücksichtigt werden müssen, um die Planbarkeit zu überprüfen. Durch die Approximation sind nun Verfahren möglich, die den approximierten Bereich nur einmal berücksichtigen müssen. Somit verschwinden für die approximierten Ereignisströme die Problematiken, die sich durch periodische Muster ergeben und die Analysen lassen sich in wesentlich geringere Zeit durchführen. Allerdings sind nicht alle Approximationen gleich sinnvoll. Es ist hierbei insbesondere von Interesse, ob die Approximation es erlaubt, den maximalen Fehler, der durch sie erzeugt werden kann, innerhalb eines akzeptablen Bereichs einzugrenzen. [9]
In [4] wird sogar ein Algorithmus zur Planbarkeitsanalyse unter EDF-Schedulern vorgestellt, der es schafft, die Taskaktivierungen genau so stark zu approximieren, wie es notwendig ist, um ein exaktes Ergebnis zu erhalten. Im statistischen Vergleich mit anderen exakten Verfahren konnte für dieses Verfahren ein erheblich geringerer Laufzeitaufwand nachgewiesen werden [13], [23].

2.3 Normierungen

Das im vorherigen Abschnitt vorgestellte Modell reicht prinzipiell aus, um korrekte Ereignisströme zu beschreiben, solange diese eine subadditive Eigenschaft haben. Durch die Flexibilität des Modells lassen sich in ihm nun aber auch gleiche Ereignismuster durch verschiedene Strukturen abbilden. Es ist daher aus verschiedenen Gründen sinnvoll, diese Möglichkeiten zu reduzieren. Insbesondere lassen sich durch bestimmte Normierungen Sonderfälle ausschließen. Dadurch können einige Operationen effizienter auf der Struktur ausgeführt werden, da diese Fälle nicht mehr berücksichtigt werden müssen.
Das ist auch das Ziel der *Separation Condition*. Es ist eine Bedingung, die die Ereignisstrommodelle einhalten sollen. Demnach erfüllt ein Ereignisstrom die Separation Condition, wenn für alle Elemente θ gilt: $T_\theta \geq l_\theta$. Diese Bedingung muss auch rekursiv für alle Unterelemente gelten.
In der Gleichung bezeichnet l_θ die Länge des kleinsten Intervalls, in dem alle Ereignisse des Musters S_θ auftreten. Der Wert für l_θ lässt sich wie folgt berech-

nen:

$$l_\theta = \begin{cases} IBF(n, \Theta') & S = \Theta' \\ nG & S = G \\ 0 & S = \infty \end{cases}$$

In der Gleichung bezeichnet $IBF(n, \Theta')$ die Intervallschranken-Funktion , die für eine Anzahl von Ereignissen die Länge des kleinsten Intervalls liefert, in dem Θ' diese Anzahl an Ereignissen erzeugen kann. Sie ist gegeben durch:

$$IBF(n, \Theta) = \min\{I \mid EBF(I, \Theta) = n\}$$

Weniger formal ausgedrückt, erlaubt es die Separation Condition nicht, dass sich die Ereignismuster zweier Instanzen desselben periodischen Elementes überlappen. Diese Bedingung kann noch strenger formuliert werden, wenn man zusätzlich noch das Überlappen von Instanzen verschiedener Elemente verbieten möchte.[1], [2]

Kapitel 3

Realtime Calculus

Der Realtime Calculus ist ein Ansatz, um Echtzeitsysteme mathematisch zu beschreiben und zu analysieren. Er hat seinen Ursprung im Network Calculus, der zur Analyse von Netzwerkverhalten dient [6]. Die Ergebnisse des Network Calculus wurden in [18, 20] und [19] aufgegriffen und auf die Schedulinganalyse von Echtzeitsystemen angewandt.

3.1 Systemabstraktion

Ein wichtiger Analyseschritt des Realtime Calculus ist die Abstraktion des Systems auf ein notwendiges und ausreichendes Maß an Informationen. Idealerweise wird die Echtzeitanalyse bereits in einem frühen Stadium der Entwicklung auf dem vorhandenen Design durchgeführt [21]. Dieses muss dazu in eine formale, mathematische Notation überführt werden. Mit fortschreitendem Entwicklungsstand können dann weitere Details der Implementierung in die Analyse einfließen. Im Folgenden soll hier gezeigt werden, wie die Abstraktion beim Realtime Calculus typischerweise aussieht.

3.1.1 Verarbeitungseinheiten und Tasks

Ein Echtzeitsystem besteht meist aus vielen verschiedenen Komponenten, wie Prozessoren, Speicher und Busse. Die Kommunikation der aktiven Komponenten kann dabei auf unterschiedliche Weisen realisiert sein. So ist der Informationsaustausch beispielsweise über gemeinsamen Speicher, Busse oder auch Interrupts möglich. Der Aufbau eines solchen Systems ist jedoch sehr stark vom Anwendungsgebiet abhängig. Aber selbst für eine bestimmte Anwendung existieren meist eine Vielzahl von verschiedenen Entwurfsmöglichkeiten. In [19] wurde zum Beispiel ein Verfahren verwendet, dass die Entwurfsraumanalyse für Netzwerkprozessoren mit Hilfe des Realtime Calculus durchführt. In dieser Arbeit ist eine Entwurfsraumanalyse aber nicht von Interesse. Der Fokus liegt hier auf der Echtzeitanalyse eines konkreten Entwurfs.
Der Realtime Calculus benötigt nur eine sehr abstrakte Beschreibung des physikalischen Entwurfs. Das hat den großen Vorteil, dass er relativ einfach für die Analyse einer Vielzahl von verschiedenen Entwürfen eingesetzt werden kann.
Bei der Echtzeitanalyse bestehen viele Analogien zwischen Bussen und Pro-

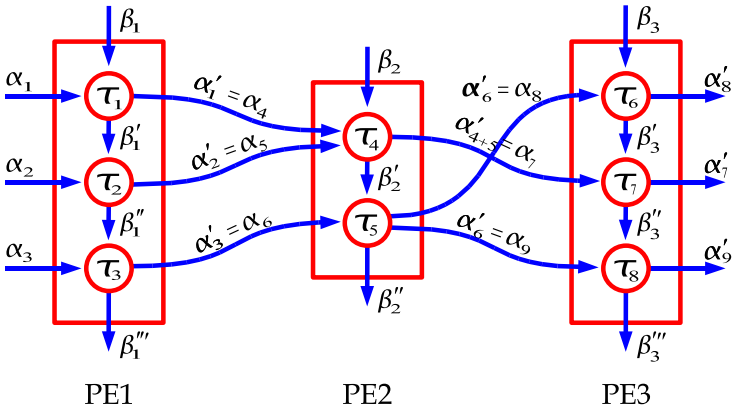
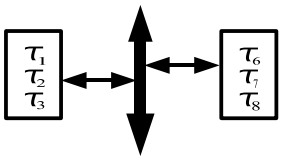


Abbildung 3.1: Systemabstraktion mit 3 Verarbeitungseinheiten

zessoren. Beide müssen bestimmte Aufgaben bewältigen, für die sie nur eine begrenzte Ressource besitzen. Bei den Prozessoren sind die Aufgaben die verschiedenen Tasks, während es bei den Bussen die Nachrichten- und Datenpakete sind. Ähnlich wie bei den verschiedenen Tasks eines Prozessors, können auch auf einem Bus Nachrichten verschiedenen Types und Dauer verschickt werden. Abhängig von der Dauer, die die Aufgaben den Prozessor oder den Bus belegen, verbrauchen sie eine gewisse Menge an Ressource des Prozessors bzw. Busses. Die Ressource des Prozessors wird dabei üblicherweise in Instruktionen pro Zeit und bei den Bussen in Datenwörter pro Zeit gemessen.
Aufgrund der bestehenden Analogien werden für die Analyse Prozessoren und Busse durch Verarbeitungseinheiten dargestellt. Ihre Aufgaben werden als Tasks bezeichnet. Da sich eine Taskaktivierung für eine Verarbeitungseinheit als Wunsch des Task darstellt, den Prozessor für eine bestimmte Zeit zu belegen, wird die Ressource der Verarbeitungseinheiten nun in der Einheit *Prozessorzeit* gemessen.
An dieser Stelle sei darauf hingewiesen, dass die Beschreibung der Tasks und Verarbeitungseinheiten nicht zum Realtime Calculus direkt gehören, da dieser ein mathematischer Kalkül ist. Er beinhaltet lediglich Regeln für die Beschreibung und Transformation der im nächsten Abschnitt vorgestellten Ankunfts- und Servicekurven. Das Modell der Tasks und Verarbeitungseinheiten gehört zur Interpretation des Kalküls. Das gleiche gilt für die Interpretation der im nächsten Abschnitt betrachteten Kurven als Ankunfts- und Servicekurven. Der Realtime Calculus ist daher ein Mittel der Echtzeitanalyse, stellt diese aber nicht selbst dar. Da jedoch die umgangssprachliche Verwendung von Kalkülen meistens ihre Interpretation implizit miteinbezieht, wird im weiteren Verlauf der Arbeit der Begriff Realtime Calculus die Interpretation und das Systemmodell miteinschließen.
In Abbildung 3.1 ist ein sehr einfaches Systemmodell zu sehen. Es besteht aus drei Verarbeitungseinheiten, die als Rechtecke dargestellt sind, und acht durch Kreise gekennzeichnete Tasks. Dieses Beispiel ist zur Illustration gedacht und bildet kein reales System ab. In der Praxis sind bedeutend komplexer Aufbauten zu erwarten.

Ein solches Modell könnte beispielsweise zwei Prozessoren beschreiben, die über einen Bus miteinander kommunizieren. Die Verarbeitungseinheiten PE1 und PE3 wären dabei die Prozessoren und PE2 der Bus.



Für die Beschreibung der Tasks im Realtime Calculus werden die Worst Case Execution Time (WCET) und die Deadline des Tasks benötigt. Ein Task wird dabei so interpretiert, dass er zur Ausführung immer die Worst Case Execution Time benötigt und jeweils nach Ablauf dieser ein Ereignis erzeugt. Dies ist eine erhebliche Einschränkung, da die Ausführungszeit realer Tasks durchaus stark variieren kann. Außerdem werden Ereignisse nicht immer am Ende der Taskausführung erzeugt. Es können durchaus mehrere Ereignisse während der Ausführung eines Tasks erzeugt werden. Hier ist sicherlich auch der größte Unterschied zwischen Tasks auf Prozessoren und Bussen zu finden. Die Annahme gleichbleibender Ausführungszeiten wird je nach verwendetem Protokoll bei Nachrichtenpaketen sicherlich am ehesten zutreffen. Außerdem erzeugen Nachrichtenpakete normalerweise tatsächlich nur ein Ereignis, und zwar wenn das Paket übertragen wurde.

3.1.2 Ankunfts- und Servicekurven

Für die Analyse fehlt nun noch eine Charakterisierung der Verarbeitungseinheiten und der Taskaktivierungen. Außerdem müssen die Abhängigkeiten der Tasks untereinander ausgedrückt werden. Diese Aspekte werden im Realtime Calculus durch das Modell der Ankunfts- und Servicekurven dargestellt, das im Folgenden beschrieben wird.
Ein Großteil dieser Informationen ist [6] und [20] entnommen. Dabei befasst sich [6] hauptsächlich mit den mathematischen Grundlagen, die in [20] für den Realtime Calculus adaptiert werden.

3.1.2.1 Ankunftskurven

Die Aktivierungen eines Tasks lassen sich für einen konkreten Lauf des Systems beobachten und mittels einer Funktion darstellen. Eine solche Funktion wird Ankunftsfunktion R genannt. $R(t)$ ist dabei die Anzahl an Ereignissen (Taskaktivierungen), die innerhalb des Intervalls $[0, t[$ auftreten [20]. Es ist leicht ersichtlich, dass eine solche Funktion monoton steigend ist. Die Ereignisse, die innerhalb eines beliebigen Intervalls $[s, t[$ mit $0 \leq s \leq t$ auftreten, lassen sich durch $R(t) - R(s)$ angeben.
Bei der Echtzeitanalyse sind jedoch keine konkreten Läufe des Systems interessant, vielmehr sollen Garantien für beliebige Läufe ermittelt werden. Daher werden obere und untere Schranken benötigt, die Taskaktivierungen beschreiben, die bei keinem möglichen Lauf des Systems über- bzw. unterschritten werden können. Solche Schranken lassen sich jedoch nicht direkt über minimale oder maximale Ankunftsfunktionen angeben, da solche nicht zwangsweise existieren müssen. Stelle man sich beispielsweise ein System vor, in dem es für einen Task nur zwei mögliche Ereignismuster gibt, deren Ankunftsfunktionen R_1 und R_2 in Abbildung 3.2 dargestellt sind. Intuitiv würde man keine dieser Funktionen als minimal oder maximal bezeichnen, da keine für den vollständigen Definiti-

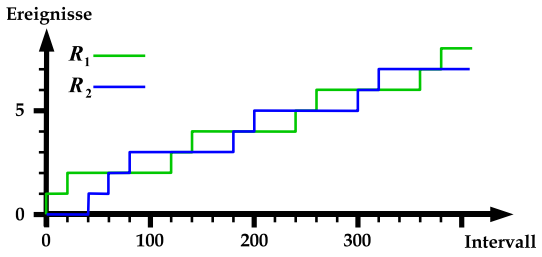


Abbildung 3.2: Zwei Ankunftskurven

onsbereich maximal bzw. minimal ist. Aber selbst wenn es immer zwei solcher Extremwertfunktionen geben würde, wäre damit nicht sicher gestellt, dass sich die kritischsten Situationen auf den Läufen des Systems ereignen, die durch diese Funktionen beschrieben werden. Es kann immer noch Läufe geben, bei denen sich die Taskaktivierungen überwiegend durchschnittlich verhalten, aber dafür in einem Zeitintervall extrem viele oder extrem wenige Ereignisse aufweisen. Jedoch entstehen gerade durch solche Zeitintervalle die kritischsten Situationen in einem Echtzeitsystem.
Die gesuchten Schranken sollten Aussagen über eben diese Intervalle machen. Dabei ist nicht so interessant, wann genau ein solches Intervall auftritt, sondern nur, wie extrem es ist, d.h. wie viele Ereignisse maximal bzw. minimal innerhalb einer bestimmten Intervalllänge auftreten.
Genau diese Funktion erfüllen die Ankunftskurven. Sie grenzen die Ereignisdichte innerhalb einer gegebenen Intervalllänge durch eine obere Ankunftskurve α^u und eine unter Ankunftskurve α^l ein, und sind definiert durch:

$$\alpha^l(t-s) \leq R(t) - R(s) \leq \alpha^u(t-s) \qquad \forall s,t : 0 \leq s \leq t \tag{3.1}$$

Anders ausgedrückt gibt $\alpha^l(\Delta)$ eine Anzahl an Ereignissen an, die innerhalb der Zeitdauer Δ sicher nicht unterschritten wird und $\alpha^u(\Delta)$ eine Anzahl, die sicher nicht überschritten wird. [20]
Nach dieser Definition kann es nun aber viele Funktionen geben, die eine solche Eigenschaft erfüllen. Wie häufig bei der Verwendung von Schranken, ist man auch hier an einer möglichst kleinen oberen und einer möglichst großen unteren Schranke interessiert.
Boudec und *Thiran* zeigten in [6] anhand eines Beispiels, dass es obere Ankunftskurven gibt, die allein durch das Wissen über diese Kurve strenger formuliert werden können. Dieses Phänomen soll auch hier durch ein ähnliches Beispiel dargestellt werden, das in Abbildung 3.3 zu sehen ist. Die Werte für die folgenden Gleichungen sind dieser Abbildung zu entnehmen. Aus der Gleichung 3.1 und den Werten von α^u folgt nun, dass immer gilt:

$$\left.\begin{array}{rcl} R(20+z) - R(z) & \leq & 1 \\ R(160+z) - R(z) & \leq & 4 \\ R(180+z) - R(z) & \leq & 8 \end{array}\right\} \quad \forall z \geq 0.$$

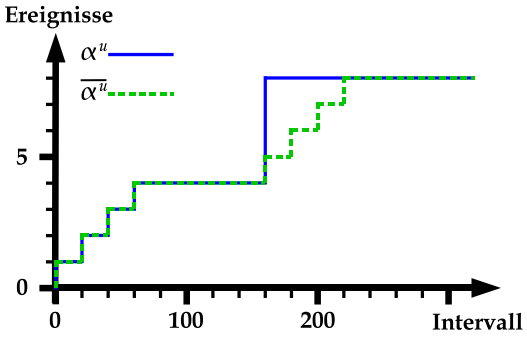


Abbildung 3.3: Ankunftskurve α^u und ihre subadditive Hülle $\overline{\alpha^u}$

Des Weiteren kann aus der Gleichung 3.1 nun abgeleitet werden, dass $\forall z \geq 0$ gilt:

$$\begin{aligned} R(180+z) - R(z) &= R(180+z) - R(z) + R(160+z) - R(160+z) \\ &= [R(180+z) - R(160+z)] + [R(160+z) - R(z)] \\ &\leq \alpha^u(20) + \alpha^u(160) \\ &= 5 \end{aligned}$$

Damit wurde nun gezeigt, dass für eine Intervalllänge von 180 der Wert $\alpha^u(20) + \alpha^u(160)$ eine bessere obere Schranke darstellt, als $\alpha^u(180)$.
Diese Beobachtung führte *Boudec* und *Thiran* zur Definition einer *guten* Ankunftskurve, die ein solches Verhalten eben nicht aufweist. Demnach ist α^u eine *gute* Ankunftskurve, wenn $\alpha^u = \overline{\alpha^u}$ gilt, wobei $\overline{\alpha^u}$ die subadditive Hülle von α^u ist.
Da man aus einer nicht guten Ankunftskurve eine kleinere obere Schranke gewinnen kann, wird für den Realtime Calculus angenommen, dass alle Ankunftskurven gut und somit subadditiv sind:

$$\alpha^u(I+J) \leq \alpha^u(I) + \alpha^u(J) \quad \forall I, J \geq 0$$

In [6] wurde diese Eigenschaft jedoch nur für die obere Schranke gezeigt, da der Network Calculus nur diese verwendet. Für die untere Schranke lässt sich über eine ähnliche Begründung auch eine ähnliche Eigenschaft einfordern. Eine gute untere Ankunftskurve sollte nun superadditiv sein, d.h. es sollte gelten:

$$\alpha^l(I+J) \geq \alpha^l(I) + \alpha^l(J) \quad \forall I, J \geq 0$$

Mithilfe der Definition der Ankunftskurven (Gleichung 3.1) lässt sich nun zeigen, warum diese Eigenschaft problemlos gefordert werden kann.
Demnach gilt $\forall I, J, z \geq 0$:

$$R(I+J+z) - R(z) \geq \alpha^l(I+J).$$

Weiterhin gilt aber auch:

$$\begin{aligned} R(I+J+z) - R(z) &= R(I+J+z) - R(z) + R(J+z) - R(J+z) \\ &= [R(I+J+z) - R(J+z)] + [R(J+z) - R(z)] \\ &\geq \alpha^l(I) + \alpha^l(J) \end{aligned}$$

Fasst man diese beiden Ungleichungen zusammen, so erhält man: $R(I+J+z)-R(z) \geq \min(\alpha^l(I)+\alpha^l(J),\ \alpha^l(I+J))$. Wenn also α^l eine untere Schranke für die Ankunftsfunktion R ist, so ist es auch die superadditive Hülle $\hat{\alpha^l}$.
Daher werden für die unteren Ankunftskurven im Realtime Calculus nur superadditive Kurven verwendet.
Damit wurde nun herausgestellt, auf welche Weise die Ankunftskurven die Taskaktivierungen im Realtime Calculus beschreiben, und welche Eigenschaften diese Kurven haben sollen. Allerdings haben die Ankunftskurven noch eine weitere Aufgabe. Sie geben die Taskabhängigkeiten untereinander an. Ein Task kann nicht nur über Ereignisse aktiviert werden, sondern selber auch Ereignismuster erzeugen, die andere Tasks aktivieren. In Abbildung 3.1 sind solche Abhängigkeiten dadurch gekennzeichnet, dass vom aktivierendem Taskknoten eine Ankunftskurve zum aktiviertem Task verläuft.

3.1.2.2 Servicekurven

Die Tasks sind nicht nur aufgrund des Kontroll- und Datenflusses von einander abhängig, sondern auch aufgrund der von ihnen gemeinsam genutzten Ressource. Da diese zu jeder Zeit immer nur von einem Task genutzt werden kann, muss ein Ablaufplan für die Tasks erstellt werden. Diese Aufgabe übernimmt der Scheduler.
Bei der Planbarkeitsanalyse von Einprozessorsystemen muss ein Algorithmus gewählt werden, der die Analyse für den gewählten Scheduler durchführen kann. Es existieren daher für jeden Scheduler eine Reihe von möglichen Algorithmen, die auf verschiedene Weisen vorgehen.
Mit dem Realtime Calculus sollen aber auch Systeme mit mehrere Prozessoren untersucht werden, auf denen unter Umständen auch verschiedenen Scheduling-Algorithmen ablaufen. Um nun möglichst viele Scheduler zu unterstützen, wird das Scheduling-Konzept im Realtime Calculus abstrahiert. Das geschieht über die sogenannten Servicekurven. Jeder Task besitzt beim Realtime Calculus eine eingehende Servicekurve β und eine ausgehende Servicekurve β' (s.Abbilung 3.1). Die eingehende Servicekurve beschreibt dabei, wie viel Prozessorzeit dem Task maximal und minimal zur Verfügung stehen. Die ausgehende beschreibt dagegen, wie viel Prozessorzeit nach Abarbeitung des Task den anderen Tasks noch bleibt. Außerdem existiert für jede Verarbeitungseinheit eine initiale Servicekurve. Sie gibt die Menge an Prozessorzeit an, die der Prozessor insgesamt zur Verfügung stellt. Mit diesen Konstrukten lassen sich nun verschiedene Scheduler flexibel darstellen.
Am einfachsten lässt sich dies sicherlich am Beispiel eines Schedulers verdeutlichen, der nach statischen Prioritäten arbeitet und die Ausführung eines Tasks unterbricht, wenn ein anderer Task mit höherer Priorität aktiviert wird. Ein solcher Scheduler wurde auch in Abbildung 3.1 für alle Verarbeitungseinheiten gewählt. Die initiale Servicekurve der Verarbeitungseinheit ist in diesem Fall zugleich die eingehende Servicekurve des Tasks mit höchster Priorität, da dieser immer sofort nach Aktivierung den Prozessor erhält. Seine ausgehende Servicekurve mit der verbleibenden Prozessorzeit ist die Zeit, die dem Tasks mit der nächst höheren Priorität zur Verfügung steht, und somit dessen eingehende Servicekurve. So werden die Tasks der Priorität nach miteinander verknüpft.
Auf ähnliche Weise lassen sich nun auch andere Scheduler modellieren. In Abbildung 3.4 sind einige Beispiele aufgeführt, die sich auch in [21] finden. Im Fall

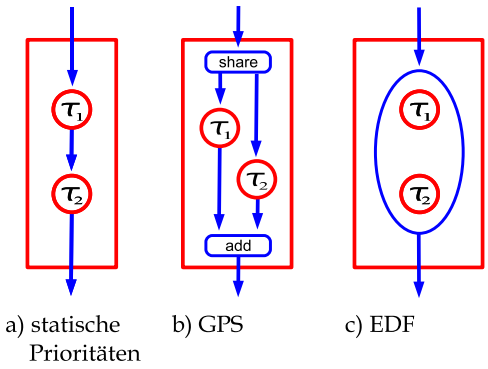


Abbildung 3.4: Beispiele verschiedener Scheduler

a) nutzt die Verarbeitungseinheit einen Scheduler, der wie oben beschrieben mit statischen Prioritäten arbeitet. Das zweite Beispiel zeigt die Verwendung eines auf GPS (Generalized Processor Sharing) basierenden Schedulers, bei dem jeder Task einen prozentualen Anteil der verfügbaren Prozessorzeit zugesichert bekommt. Solche Scheduler werden zum Beispiel in Netzwerken verwendet, in denen Anwendungen einen Mindestdatendurchsatz benötigen. Bei diesem Scheduler muss aus der initiale Servicekurve der Verarbeitungseinheit in einem zusätzlichen Schritt für jeden Task eine eingehende Servicekurve erstellt werden, die dessen Anteil an der Gesamtprozessorzeit beschreibt. Um abschließend die verbleibende Servicekurve der Verarbeitungseinheit zu bestimmen, müssen die ausgehenden Servicekurven der Tasks erst wieder zu einer Servicekurve kombiniert werden.
Schwieriger wird es allerdings bei Schedulingverfahren wie EDF, bei denen die Antwortzeit jedes Tasks von der Ausführung aller anderen abhängt. Hier stößt die Flexibilität des Realtime Calculus an seine Grenzen, da die eingehende Servicekurve eines Task aus den ausgehenden aller anderen erstellt werden muss. Das führt zu Zyklen im Graphen, der die Taskabhängigkeiten abbildet, die mit dem Realtime Calculus nicht direkt gehandhabt werden können. Daher müssen für die Bestimmung der ausgehenden Kurven bei solchen Schedulern jeweils angepasste Verfahren verwendet werden.

Der Aufbau und die Definition einer Servicekurve ähnelt sehr stark den Ankunftskurven. Die Beschreibung der verfügbaren Prozessorzeit erfolgt auch über eine obere und eine untere Schranke. Sie geben für eine Intervalllänge die maximale und minimale Menge an Prozessorzeit an, die ein Task während dieser Zeitdauer erhalten kann. Das bedeutet, dass auch durch diese Kurven nicht das genaue zeitlich Verhalten beschrieben wird, sondern Aussagen über besonders extreme Situationen gemacht werden.
Analog zur Definition der Ankunftskurven, erfolgt die Definition der Servicekurven über alle zur Laufzeit möglichen Szenarien des Systems. Die verfügbare Prozessorzeit für einen konkreten Ablauf des Systems lässt sich über die Servicefunktion C angeben. Dabei gibt $C(t)$ die Prozessorzeit an, die während des Intervalls $[0, t[$ einem Task zur Verfügung steht. Für ein beliebiges Intervall $[s, t[$ mit $0 \leq s \leq t$ lässt sich der Wert durch $C(t) - C(s)$ bestimmen.

Mit der Servicefunktion lassen sich nun die obere Schranke β^u und die untere Schranke β^l der Servicekurve wie folgt definieren:

$$\beta^l(t-s) \leq C(t) - C(s) \leq \beta^u(t-s) \qquad \forall s,t : 0 \leq s \leq t$$

Wie auch bei den Ankunftskurven ist man hier an möglichst engen Schranken interessiert. Die Begründung der Definition einer *guten* Ankunftskurve erfolgte über die Definition der Ankunftskurven selbst. Aufgrund der Symmetrie in den Definitionen der Ankunfts- und Servicekurven, lässt sich die Bezeichnung *gut* auch auf die Servicekurven anwenden. Das bedeutet insbesondere, dass für die Verwendung im Realtime Calculus davon ausgegangen werden kann, dass β^l immer eine superadditive und β^u eine subadditive Funktion darstellen.
Für die initiale Servicekurve einer Verarbeitungseinheit wird meistens eine Gerade durch den Ursprung mit Steigung eins verwendet, die eine idealisierte Sicht der Prozessorzeit zeigt, bei der der Prozessor zu jeder Zeit maximale Leistung bringt.

Damit wurde nun beschrieben, wie ein Systemmodell aussehen muss, um es für die Analyse mit dem Realtime Calculus verwenden zu können. Bei den Ankunfts- und Servicekurven wurde deutlich, dass diese mathematisch identisch definiert wurden und sich lediglich in ihrer Interpretation unterscheiden. Sie bilden die zentralen Elemente für die Transformationsregeln des eigentlichen Calculus.

3.2 Calculus

Wie bereits im vorherigen Abschnitt erwähnt, besteht der Kern des Realtime Calculus hauptsächlich aus Transformationsregel. Diese Regeln beziehen sich immer auf einen Taskknoten und geben an, wie aus den eingehenden Ankunfts- und Servicekurven die ausgehenden errechnet werden können (s. Abbildung 3.5).

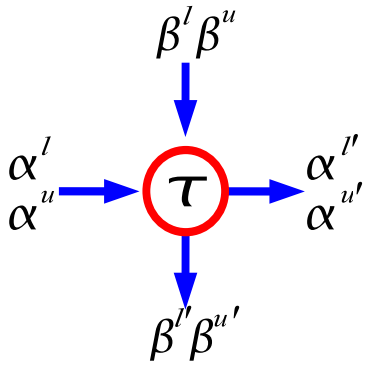


Abbildung 3.5: Taskknoten

3.2.1 Vorbereitende Operationen

Abbildung 3.5 zeigt die Darstellung eines Taskknoten. Dabei hat ein Task genau eine eingehende und eine ausgehende Ankunftskurve. Das ist auch eine Annah-

me bei den Transformationsregeln des Realtime Calculus. Zu Beginn des Kapitels wurde jedoch in Abbildung 3.1 ein System gezeigt mit Tasks, die mehrere eingehende oder ausgehende Ankunftskurven besitzen. Um den Anforderungen gerecht zu werden, müssen daher die eingehenden Ankunftskurven zusammengefasst werden.

3.2.1.1 Summation der eingehenden Ereignisse

Gesucht sind bei der Zusammenfassung Schranken, die für eine Intervalllänge angeben, wie viele Ereignisse minimal und maximal durch alle eingehenden Ankunftskurven zusammen beschrieben werden können. Es ist leicht ersichtlich, dass solche Schranken einfach über die Addition aller Kurven erstellt werden können. Allerdings sind diese Grenzen unter Umständen nicht die kleinsten bzw. größten Schranken, selbst wenn dies für alle Summanden gilt. Das tritt auf, wenn für eine gegebene Intervalllänge das kritischste Intervall einer eingehenden Ankunftskurve im realen System nie zeitgleich mit dem kritischsten Intervall einer anderen eingehenden Ankunftskurve zusammenfällt.
Im Bereich der Echtzeitanalyse werden solche Eigenschaft jedoch meistens toleriert. So beruhen Verfahren zur Analyse periodischer Tasksysteme oftmals auf der Annahme, dass es sich dabei um ein synchrones Taskset handelt, bei dem die erste Aktivierung aller Tasks synchron zum Zeitpunkt Null stattfindet. Das reale System kann durch die Offsets der Tasks jedoch so aufgebaut sein, dass nie alle Tasks gleichzeitig aktiviert werden.
Für den Realtime Calculus muss man diese weicheren Grenzen akzeptieren, da die Ankunftskurven keine Informationen mehr über das genaue zeitliche Auftreten der kritischen Intervalle enthalten.
Bei dieser Vorgehensweise bleibt die Sub- bzw. Superadditivität erhalten. Denn die Summe der Ankunftskurven ist immer noch eine *gute* Kurve, wenn das auch alle Summanden sind. Diese Eigenschaft lässt sich für die Addition zweier guter Ankunftskurven α_1^u und α_2^u leicht zeigen. Hierbei ist die Summe $\alpha_1^u + \alpha_2^u$ eine gute Kurve, da sich aus

$$\alpha_1^u(I+J) \leq \alpha_1^u(I) + \alpha_1^u(J) \quad \text{und} \quad \alpha_2^u(I+J) \leq \alpha_2^u(I) + \alpha_2^u(J)$$

sehr einfach

$$\alpha_1^u(I+J) + \alpha_2^u(I+J) \leq \alpha_1^u(I) + \alpha_1^u(J) + \alpha_2^u(I) + \alpha_2^u(J)$$

folgern lässt. Das gleiche gilt analog für die unteren Ankunftskurven und lässt sich iterativ auf eine beliebige Anzahl von Summanden erweitern.

3.2.1.2 Skalierung

Da die Einheit der Servicekurve Rechenzeit ist und die Ankunftskurve die Anzahl an Ereignissen angibt, besteht noch kein direkter Zusammenhang zwischen beiden Kurven. Dieser wird über die Ausführungszeit des Tasks hergestellt. Jedes Ereignis der Ankunftskurven stellt eine Taskaktivierung dar. Das bedeutet, dass vom Prozessor so viel Rechenzeit zur Verfügung gestellt werden muss, wie der Task zur Ausführung benötigt. Daher wird die eingehende Ankunftskurve mit der Ausführungszeit des Tasks multipliziert. Die resultierende Kurve stellt

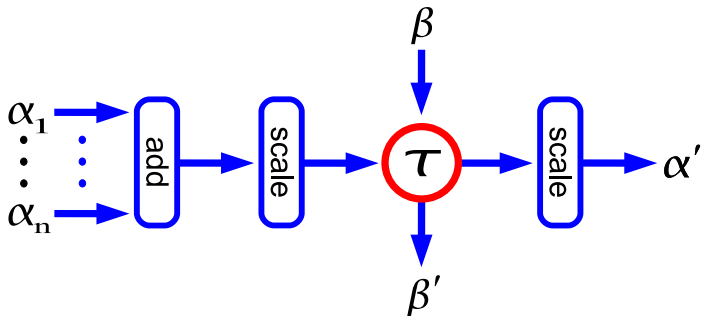


Abbildung 3.6: Vorbereitende und nachbereitende Operationen

nun die minimal und maximal vom Task angeforderte Rechenzeit dar.
Als Folge dessen geben die ausgehenden Ankunftskurven an, wie viel Prozessorzeit dieser Task innerhalb eines Zeitintervalls minimal und maximal beansprucht. Um hieraus die Anzahl der Ereignisse zu bestimmen, die der Task erzeugt, muss die ausgehende Ankunftskurve wiederum mit dem Kehrwert der Ausführungszeit skaliert werden.[3]
Die genaueren Zusammenhänge werden im nächsten Abschnitt erläutert.
Der Ablauf der hier beschriebenen, vorbereitenden Operationen ist für einen Taskknoten in Abbildung 3.6 zu sehen.

3.2.2 Transformationsregeln

Der zentrale Punkt des Realtime Calculus ist die Berechnung der ausgehenden Kurven eines Taskknoten. Im Folgendem werden nun die Transformationsregeln beschrieben, die dies leisten. Außerdem wird versucht eine möglichst intuitive Begründung dieser Regeln zu geben.

3.2.2.1 Untere Servicekurve

$$\beta^{l'}(\Delta) = \sup_{0 \le t \le \Delta} \{\beta^l(t) - \alpha^u(t)\} \tag{3.2}$$

Die ausgehende untere Schranke der Servicekurve soll beschreiben, wie viel Prozessorzeit der Task nachfolgenden Tasks in jedem Fall zur Verfügung lässt. Dies entspricht also der Prozessorzeit, die in den kritischsten Situationen noch zugesichert werden kann.
Eine solche Situation tritt immer dann ein, wenn der Task selber nur sehr wenig Prozessorzeit erhält, aber gleichzeitig sehr viel anfordern möchte. Die Beschreibung von *sehr wenig Prozessorzeit erhalten* erfolgt eben durch die untere Servicekurve und die Beschreibung von *sehr viel Prozessorzeit anfordern* durch die obere Ankunftskurve. Die Idee ist also, dass man von der knappen Prozessorzeit (β^l) das Maximum an Rechenzeitanforderung (α^u) abzieht.
Der Term $\beta^l(t) - \alpha^u(t)$ ist jedoch durch eine Supremumfunktion geklammert. Die Begründung hierfür lässt sich gut anhand von Abbildung 3.7 zeigen. Im

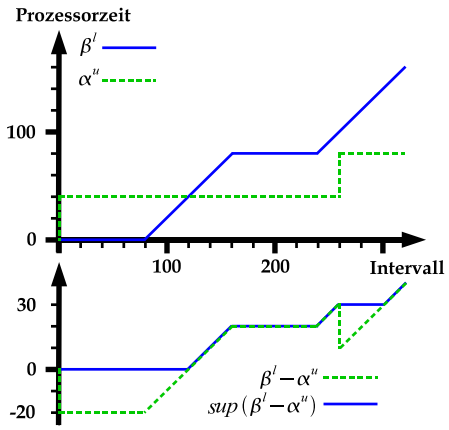


Abbildung 3.7: Berechnung der unteren Servicekurve.
oben: Obere Ankunfts- und untere Servicekurve.
unten: Differenz- und Supremumbildung

oberen Koordinatensystem sind β^l und α^u abgebildet. Im unteren gibt der gestrichelte Graph die Differenz beider Kurven an. Man erkennt sofort, dass die Differenz die formalen Anforderungen einer Servicekurve nicht erfüllt, da sie nicht monoton steigend ist.
Die Differenz ist aber auch nicht das Ergebnis, das man für die ausgehende Servicekurve erwarten würde. Das ist im Beispiel gut zum Zeitpunkt 260 ersichtlich. Dort fordert der Task 40 Einheiten Prozessorzeit an. Durch die Differenzbildung wird diese Menge direkt von der eingehenden Servicekurve abgezogen. In der Realität führt eine solche Anforderung jedoch dazu, dass die nächsten 40 Einheiten, die dem Task zur Verfügung gestellt werden, auch von ihm verbraucht werden, und somit den nachfolgenden Tasks nicht zur Verfügung stehen. Daher erwartet man, dass die resultierende Kurve an diesem Punkt eine Steigung von Null hat, bis die 40 Einheiten dem Task zur Verfügung gestellt wurden. Ein solches Verhalten wird über die Supremumfunktion erzielt.
Wie auch bei der Addition in Abschnitt 3.2.1.1 beruhen diese Überlegungen darauf, dass zwei kritische Situationen zeitnah auftreten. Zum Einen wird der Task sehr häufig aktiviert und zum Anderen steht wenig Prozessorzeit zur Verfügung. Da aber auch hier eine solche Kombination von Situationen im realen System unter Umständen gar nicht auftreten kann, ist die errechnete Servicekurve nicht unbedingt die optimale untere Schranke. Das kann jedoch an dieser Stelle und bei den nachfolgenden Operationen akzeptiert werden.

3.2.2.2 Obere Servicekurve

$$\beta^{u'}(\Delta) = \sup_{0 \leq t \leq \Delta} \{\beta^u(t) - \alpha^l(t)\} \tag{3.3}$$

Die Überlegungen zur Berechnung der oberen Servicekurve sind analog zu denen bei der unteren Servicekurve. Hier ist nun eine obere Schranke an Prozessorzeit gefragt, die dieser Task nachfolgenden Task im Besten Fall überlässt. Das tritt eben dann auf, wenn der Tasks sehr viel Prozessorzeit erhält, aber nur wenig davon anfordern möchte. Daher wird bei der oberen Servicekurve die Differenz aus β^u und α^l gebildet. Die weiteren Überlegungen decken sich mit denen im vorherigen Abschnitt.

3.2.2.3 Untere Ankunftskurve

$$\alpha^{l'}(\Delta) = \inf_{0 \leq t \leq \Delta} \{\alpha^l(t) + \beta^l(\Delta - t)\} \tag{3.4}$$

Bei den ausgehenden Ankunftskurven ist die Annahme, dass der Task eine konstante Ausführungszeit hat und immer am Ende seiner Ausführung ein Ereignis erzeugt. Um nun eine Schranke für die minimal von diesem Task erzeugten Ereignisse zu erhalten, muss man annehmen, dass der Task selber nur sehr selten aktiviert wird (α^l), da er dann auch nur wenige Ereignisse erzeugen kann. Diese Situation kann noch extremer werden, wenn der Task noch weniger Prozessorzeit erhält (β^l), als er minimal anfordern möchte. Aus den möglichen Kombinationen dieser beiden Situationen ($\alpha^l(t) + \beta^l(\Delta - t)$), wird nun diejenige heraus gesucht, die insgesamt nicht länger als Δ andauert und in der Summe am wenigsten Ereignisse erzeugt. Dafür wird die Infimumfunktion benötigt.

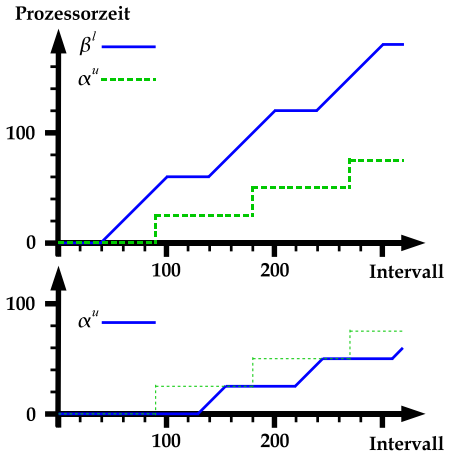


Abbildung 3.8: Berechnung der unteren Ankunftskurve.
oben: Untere Ankunfts- und Servicekurve.
unten: Ausgehende untere Ankunftskurve

In der Funktionsbeschreibung dient t im Prinzip nur als Laufvariabel, um alle möglichen Kombinationen aus α^lund β^lzu beschreiben. Die beiden Kurven sind in der Berechnung jedoch gleichberechtigt. Das heißt, dass $\inf_{0\leq t\leq\Delta}\{\alpha^l(\Delta-t)+\beta^l(t)\}$ das selbe Ergebnis liefert.
In Abbildung 3.8 ist das Ergebnis einer solche Berechnung beispielhaft dargestellt.

3.2.2.4 Obere Ankunftskurve

$$\alpha^{u'}(\Delta) = \inf_{0\leq t\leq\Delta}\{\sup_{v\geq 0}\{\alpha^u(t+v)-\beta^l(v)\}+\beta^u(\Delta-t)\} \tag{3.5}$$

Die Transformationsregel für die obere Ankunftskurve wirkt sehr kompliziert. Sie weist jedoch eine gewisse Analogie zur Regel für die untere Ankunftskurve auf. Ersetzt man den Term $\sup_{v\geq 0}\{\alpha^u(t+v)-\beta^l(v)\}$ durch $\alpha^u(t)$ so erhält man

$$\alpha^{u'}(\Delta) = \inf_{0\leq t\leq\Delta}\{\alpha^u(t)+\beta^u(\Delta-t)\} \tag{3.6}$$

Hier finden sich nun die gleichen Symmetrien zur unteren Ankunftskurve wie bei den Transformationsregeln der Servicekurven. Die Interpretation von Gleichung 3.6 ist nun wie bei der unteren Ankunftskurve. Es wird eine Kombination von Situationen gesucht, in denen sehr viele Taskaktivierungen (α^u) stattfinden und gleichzeitig sehr viel Prozessorzeit zur Verfügung gestellt wird, um diese Aktivierungen alle zu bearbeiten (β^u). Die Grenzen der Infimumfunktion sorgen auch hier dafür, dass die Gesamtlänge dieser Situationen nicht länger als die betrachtete Intervalllänge Δ ist. Durch die Infimumfunktion selbst ist gewährleistet, dass der errechnete Wert nicht größer ist als die maximal zur Verfügung stehende Prozessorzeit.
Für die obere Ankunftskurve lassen sich unter Umständen jedoch Situationen finden, in denen noch mehr Ereignisse erzeugt werden. Eine solche Situation tritt auf, wenn sich vor einem betrachteten Intervall der Länge Δ Rechenzeitanforderungen aufgestaut haben, die noch nicht bearbeitet werden konnten. Wenn diese zusätzlich zu den Anforderungen, die innerhalb des betrachteten Intervalls auftreten, bearbeitet werden können, so werden innerhalb des Intervalls auch mehr Ereignisse erzeugt, als die Gleichung 3.6 errechnet hat.
Die größtmögliche Menge an aufgestauten Ereignissen lässt sich nun über die Funktion $\sup_{v\geq 0}\{\alpha^u(v)-\beta^l(v)\}$ errechnen. Kombiniert man dies mit $\alpha^u(t)$, so erhält man den Term aus Gleichung 3.5, der eben eine große Menge von angestauten Ereignissen gefolgt von möglichst vielen Ereignissen innerhalb eines Intervalls Δ angibt.

3.3 Einschränkungen

In Abschnitt 3.1.2.2 wurde gezeigt, wie verschiedene Scheduling-Strategien im Realtime Calculus verwendet werden können. Am einfachsten lässt sich dabei ein Scheduler handhaben, der mit statischen Prioritäten arbeitet. Andere Verfahren, wie zum Beispiel GPS, benötigen zusätzliche Operationen. Bei EDF besteht dagegen das Problem, dass für jeden Task die zur Verfügung stehende Rechenzeit von allen anderen Tasks abhängt und daher durch die Servicekurven

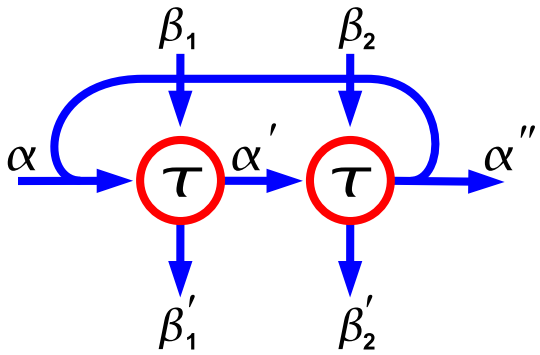


Abbildung 3.9: Zyklus im Taskgraphen

zyklische Graphen gebildet würden.
Die Behandlung von zyklischen Graphen wirft jedoch weiterer Probleme auf. Da sich die Transformationsregeln des Realtime Calculus immer nur auf einen Taskknoten beziehen, würde ein zyklischer Taskgraph dazu führen, dass die Berechnung der Kurven in einer Endlosschleife endet. Um diese Problematik zu beheben, sind sicherlich verschiedene Ansätzen denkbar. Diese erfordern aber eine ganze Reihe weiterer Überlegungen. Diese Arbeit befasst sich jedoch hauptsächlich mit den Algorithmen zur Berechnung der Tasktransformationen. Daher wird der Fokus hier auf Verarbeitungseinheiten gelegt, deren Tasks nach statischen Prioritäten geplant werden.
Zyklen im Taskgraphen können allerdings auch durch die Ankunftskurven entstehen, wie es in Abbildung 3.9 zu sehen ist. Um die beschriebene Problematik zu umgehen, werden daher in dieser Arbeit solche Taskabhängigkeiten ebenfalls ausgeschlossen.

Kapitel 4

Realtime Calculus mit hierarchischen Ereignisströmen

In Kapitel 2 wurde dargestellt, wie es mit Ereignisströmen möglich ist, kritische Häufungen von Taskaktivierungen zu beschreiben. Zudem wurde gezeigt, wie aus einem Ereignisstrom die Ereignisschranken-Funktion gewonnen werden kann, die den Ereignisstrom als subadditive, mathematische Funktion repräsentiert.
In Kapitel 3 wurde der Realtime Calculus erläutert, dessen Ankunftskurven ebenfalls Taskaktivierungen beschreiben. Für die obere Grenze der Ankunftskurven fordert er eben die subadditive Eigenschaft, die die Ereignisströme bieten. Da die Ankunftskurven lediglich auf einer mathematischen Definition beruhen, liegt es an dieser Stelle nahe, für eine konkrete Implementierung der oberen Ankunftskurve das beschriebene Ereignisstrommodell zu verwenden.
In diesem Kapitel wird nun erläutert, wie eine solche Implementierung vorgenommen werden kann. Dabei werden Algorithmen vorgestellt, die in [2] entwickelt wurden, und es ermöglichen die Berechnungen des Realtime Calculus effizient mit dem Ereignisstrommodell durchzuführen.

4.1 Beschreibung der Ankunfts- und Servicekurven

Mit dem Ereignistrommodell lassen sich nun die oberen Ankunftskurven hervorragend darstellen. Wie sieht es jedoch mit den anderen Kurven aus? Für die obere Servicekurve wird ebenfalls eine subadditive Funktion benötigt. Die Servicekurve beschreibt aber keine Ereignisse, sondern die Menge an vorhandener Prozessorzeit. Da es dem Ereignisstrommodell möglich ist, Ereignisse durch linear steigende Segmente zu approximieren, ist es ihnen auch möglich die Servicekurven abzubilden, indem die kontinuierlich nachgelieferte Ressource Prozessorzeit eben durch solche kontinuierlich steigenden Segmente dargestellt wird.
Bei den unteren Grenzen der Ankunfts- und Servicekurven ist die Darstellung durch das in Kapitel 2 erläuterte Ereignisstrommodell nicht direkt möglich. Die

untere Ankunftskurve soll eine untere Grenze für die Ereignisdichte geben und ist nicht subadditiv, sonder superadditiv. Für die Verwendung im Realtime Calculus muss daher die Definition der Ereignisströme erweitert werden. Ereignisströme können demnach entweder superadditiv oder subadditiv sein. Auf diese Weise ist es möglich auch untere Ankunfts- und Servicekurven zu beschreiben.
Bei den Servicekurven kann es vorkommen, dass die Kurve beide Eigenschaften besitzt. Dies ist typischerweise der Fall bei der initialen Servicekurve einer Verarbeitungseinheit. Sie wird durch eine Gerade der Steigung 1 dargestellt und daher gilt für sie sowohl

$$\beta(I+J) \leq \beta(I) + \beta(J) \text{ , als auch } \beta(I+J) \geq \beta(I) + \beta(J).$$

Dagegen werden sich bei den Ankunftskurven normalerweise beide Fälle ausschließen.

4.2 Problematik

Der Realtime Calculus beschreibt mathematisch, wie man aus den eingehenden Ankunfts- und Servicekurven eines Taskknotens die ausgehenden Kurven berechnet. Wenn nun die eingehenden Kurven durch das Ereignisstrommodell beschrieben werden, so möchte man natürlich, dass die ausgehenden auch durch dieses Modell beschrieben werden.
Prinzipiell ist das auch möglich, jedoch wirft dies Probleme auf, die leicht am Beispiel folgender Ereignisströme zu erkennen sind.

$$\alpha^u = \big(\big(6,0,2,((2,0,1,\infty))\big)\big)$$
$$\beta^l = \big((7,4,3,1)\big)$$

Berechnet man aus diesen Strömen mit der Gleichung 3.2 nun die ausgehende untere Servicekurve, so erhält man den Ereignisstrom, der in Abbildung 4.1 dargestellt ist. Er kann beispielsweise durch:

$$\begin{aligned}\beta^{l'} \quad = \quad & \big((42,3,3,1),(42,9,3,1),(42,13,1,1),(42,16,2,1),(42,19,1,1),\\ & (42,22,2,1),(42,25,1,1),(42,27,1,1),(42,29,1,1),(42,31,1,1),\\ & (42,33,2,1),(42,37,1,1),(42,39,3,3)\big)\end{aligned}$$

abgebildet werden. Hier tritt nun ein typisches Problem bei der Echtzeitanalyse von periodischen Tasksystemen auf . Die Periode von $\beta^{l'}$ ist wesentlich größer, als die von α^u und β^l. Sie wird aus dem kleinsten gemeinsamen Vielfachen der eingehende Kurven gebildet. Da $\beta^{l'}$ wiederum Eingabe eines anderen Tasks sein kann, kann sich die Periode bei den nachfolgenden Tasks sehr schnell immer weiter vergrößern.
Mit der Größe der Periode steigt normalerweise auch die Anzahl an Elementen, die nötig sind, um eine Periode zu beschreiben. Außerdem ist bei diesen Perioden meistens eine effiziente Darstellung mithilfe eines hierarchischen Aufbaus nur schwer möglich.
In diesem Beispiel besitzt α^u zwei Hierarchieebenen. Bei der Berechnung von $\beta^{l'}$ muss im Prinzip aber jedes Element einzeln behandelt werden. Daher findet sich diese Hierarchie bei $\beta^{l'}$ nicht wieder. Die hierarchische Darstellung ist für die Berechnung also nutzlos.

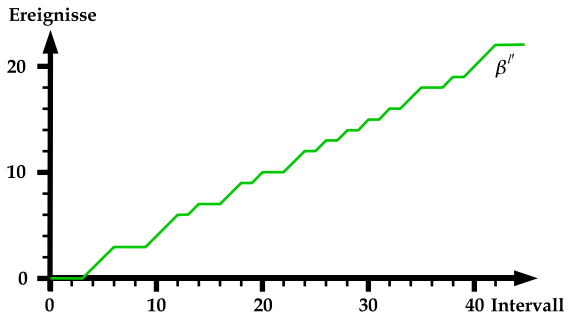


Abbildung 4.1: Vergrößerung der Periode bei Berechnung der unteren Servicekurve

Der Problematik, dass sehr große Perioden entstehen, kann durch Approximation der Kurven begegnet werden. Die Grenzen der Ankunfts- und Servicekurven sollen eine Garantie für die maximale und minimale Ereignisdichte bzw. verfügbare Prozessorzeit beschreiben. Dabei ist nicht vorgegeben, dass es sich hierbei um die kleinste obere, bzw. größte untere Schranke handeln muss. Daher ist es möglich ungefähre Schranken anzugeben, die die geforderte Garantie nicht verletzen.
In [2] wurden auf diese Weise Methoden vorgestellt, die es ermöglichen, das Ereignisstrommodell effizient mit dem Realtime Calculus zu verknüpfen. In Abschnitt 2.2 wurde das Vorgehen bei der Approximation bereits erläutert. Es kann auch auf die untere Ankunftskurve angewendet werden. Hier muss entgegen der Approximation der oberen Grenze darauf geachtet werden, dass die Ereignisschranken-Funktion der Approximation in keinem Fall einen größeren Wert als die Ereignisschranken-Funktion des Originals liefert. In Abbildung 4.2 ist die Approximation nach drei Perioden dargestellt für beide Grenzen der Ankunftskurve eines Tasks, der periodisch alle 80 Zeiteinheiten aktiviert wird.
Durch die Verwendung approximierter Ereignisströme im Realtime Calculus entstehen durch die Berechnungen nun weitere Ströme, für die weder ein hierarchischer Aufbau noch die Verwendung von Perioden hilfreich ist. Daher bietet es sich für die Berechnung an, eine etwas schlankere Datenstruktur zu verwenden, mit der verschiedene Operationen effizient durchgeführt werden können.

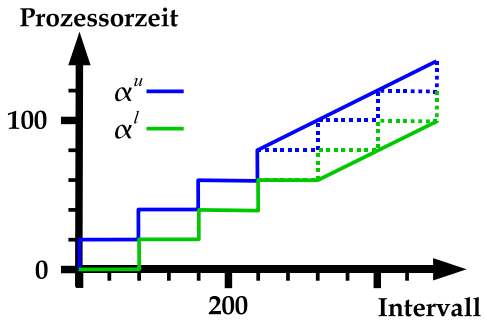


Abbildung 4.2: Approximation nach 3 Perioden

4.3 Darstellung als Testliste

In [3] nutzen *Albers*, *Bodmann* und *Slomka* bereits Testlisten als Repräsentation der Ereignisströme. Die Elemente einer solchen Testliste beschreiben dabei jeweils das Verhalten des Ereignisstroms für eine bestimmte Intervalllänge. In die Testliste werden jedoch nur Elemente für Intervalllängen eingefügt, bei denen sich das Verhalten des Stroms ändert. Jedes Element der Liste besteht dabei aus drei Parametern.

$$\tau = (a, c, G)$$

Die Intervalllänge, für die das Element das Verhalten beschreibt, wird im ersten Parameter a angegeben. Eine Testliste enthält für jede Intervalllänge immer nur maximal ein Element. Die Menge an Ereignissen bzw. an angeforderter Prozessorzeit wird mittels c beschrieben. Dieser Parameter beschreibt also eine der Stufen der Ereignisschranken-Funktion , die zum Beispiel in der Abbildung 4.2 zu sehen sind. Der letzte Parameter G schließlich dient zur Beschreibung kontinuierlich steigender Abschnitte, die bei den Servicekurven und approximierten Ankunftskurven auftreten. Dabei wird im Gradienten nicht die aktuelle Steigung der Kurve gespeichert, sondern immer nur die Änderung, die bei der Intervalllänge a auftritt. Ein sehr einfaches Beispiel ist in Abbildung 4.3 zu sehen.
In [2] wurde eine Vorgehensweise beschrieben, die aus einem approximierten Ereignisstrom eine Testliste erstellt, in der dieser approximierte Strom vollständig abgebildet wird. Anschließend wurden die Algorithmen vorgestellt, die für diese approximierten Testlisten die Berechnungen des Realtime Calculus realisieren.
Das Ziel dieser Arbeit ist es nun diese, Algorithmen in Java zu implementieren und anschließend eine Komplexitätsanalyse vorzunehmen. Eine weitere Idee bestand darin, die Algorithmen so zu erweitern, dass sowohl eine Analyse von approximierten als auch von nicht approximierten Ereignisströmen möglich ist. Dabei sollten die ursprünglichen Algorithmen nach Möglichkeit nicht verändert, sondern in vor- und nachbereitende Operationen eingebettet werden. Diese sollten die Testlisten so anpassen, dass als Resultat schließlich eine exakte Testliste berechnet wird.
Es ergaben sich dabei jedoch mehr Schwierigkeiten als ursprünglich erwartet. Dennoch wird im weiteren das Vorgehen für approximierte und exakte Testlisten erläutert. In diesem Zusammenhang soll dann gezeigt werden, an welchen Hürden es scheiterte und was getan werden müsste, um diese zu überwinden.

(2 , 0 , 1)
(3 , 0 , -1)
(4 , 1 , 0)
(5 , 1 , 0.5)
(7 , 0 , -0.5)

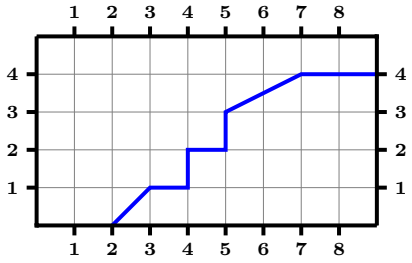


Abbildung 4.3: Testliste mit 5 Elementen und daraus resultierende Kurve

Wenn man einen nicht approximierten Ereignisstrom als Testliste darstellen möchte, stellt sich als erstes die Frage, wie lang eine solche Liste mindestens sein muss. Betrachtet man nur approximierte Ereignisströme, so kann sie sehr einfach beantwortet werden. Diese Ereignisströme besitzen keinen unendlich periodischen Teil mehr. Sie sind jeweils nur zu Beginn detailliert beschrieben und besitzen für große Intervalle nur noch einen einfachen Gradienten. Daher lässt sich für diese Ströme auch eine endliche Testlistenrepräsentation finden, die exakt den approximierten Ereignisstrom abbildet.
Um nun aber auch mit nicht approximierten Ereignisströmen arbeiten zu können, muss die Testlistenstruktur erweitert werden. Zum einen muss die Periodenlänge bekannt sein, um aus einer endlichen Testliste die Werte für große Intervalle bestimmen zu können. Im Normalfall werden aber auch Ereignisströme mit periodischen Elementen nicht ein rein periodisches Verhalten aufweisen. Das liegt einerseits daran, dass als Eingabe für das zu untersuchende System auch Ereignisströme verwendet werden können, die neben periodischen Elementen auch nicht periodische enthalten. Aber selbst wenn dies nicht der Fall ist, werden durch die Berechnungen des Realtime Calculus auch Ereignisströme gebildet, die periodische und aperiodische Bereiche besitzen. Dies ist später an einem Beispiel in Abschnitt 4.5.1.2 zu sehen. Da ein Ereignisstrom aber nicht unendlich viele aperiodische Elemente enthalten kann, kann eine Intervallgröße angegeben werden, ab der sich das Verhalten des Ereignisstroms periodisch wiederholt. Auch diese Grenze muss in der erweiterten Testlistenstruktur vorhanden sein.
Diese Grenze wird bei den nachfolgenden Überlegungen immer so interpretiert, dass Testlistenelemente, die genau den Offset dieser Grenze haben immer zum aperiodischen Bereich gezählt werden. Diese Wahl erleichtert an einigen Stellen die Implementierung. Allerdings tritt nun das Problem auf, dass bei Ereignisströmen, die gar keinen aperiodischen Bereich besitzen, das Testlistenelement mit dem Offset 0 nicht zum periodischen Bereich gehört. Daher erhält die Testliste zusätzlich ein Flag, das angibt, ob ein aperiodischer Bereich existiert und somit das Element mit Offset 0 zum periodischen oder aperiodischen Bereich gezählt wird.
Bei nicht periodischen Ereignisströmen ist der periodische Bereich leer und die Periode erhält den Wert ∞.
In der Abbildung 4.4 ist eine solche zweigeteilte Testliste mit der dazugehörigen Kurve zu sehen. Die Testliste besitzt einen aperiodischen Teil (AP), der sich bis zu einer Intervalllänge von drei erstreckt. Darauf folgt der periodische Bereich

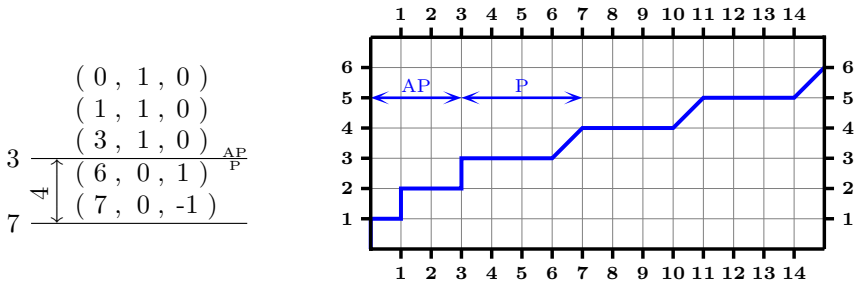


Abbildung 4.4: Testliste mit periodischem und aperiodischem Bereich

(P), der sich mit einer Periode von vier bis zur Intervalllänge sieben erstreckt. Man erkennt hier sehr gut, dass es auf diese Weise mit wenigen Testlisten-Elementen möglich ist, die Kurve unendlich fortzuführen.

4.4 Konvertierung der Testlisten

Durch die periodischen Testlisten ist es möglich, die Kurven des Realtime Calculus äquivalent durch Ereignisströme und periodische Testlisten zu beschreiben. Daher sind nun Konvertierungsverfahren gefragt, die eine Beschreibungsform in die jeweils andere übersetzen können.
Albers et al. und stellten in [2] bereits ein Verfahren vor, um aus einem hierarchischen Ereignisstrommodell eine Testliste zu generieren, die eine Periode des Stroms exakt beschreibt. Dieser Algorithmus kann auch für die Erstellung der oben vorgestellten, zweigeteilten Testlisten verwendet werden, da die zusätzlichen Informationen für die Periodenlänge und den Beginn des periodischen Bereichs sehr einfach aus dem Ereignisstrommodell abgelesen werden können.
Umgekehrt ist es auch möglich, aus einer Testliste wieder einen Ereignisstrom zu generieren. Bei der Generierung der Testlisten gehen jedoch die Informationen des hierarchischen Aufbaus des Ereignisstroms verloren. Daher können die Ereignisströme, die aus Testlisten gewonnen werden, auch nur bedingt die Möglichkeiten des hierarchischen Aufbaus verwenden. Da der hierarchische Aufbau aber in vielen Fällen die Darstellung enorm komprimiert, ist damit zu rechnen, dass die aus Testlisten gewonnenen Ereignisströme einen großen Platzbedarf haben und unübersichtlich dargestellt sind.
Die Erstellung eines Ereignisstroms aus einer Testliste erweist sich jedoch als sehr einfach. Zuerst wird der aperiodische Bereich der Testliste behandelt. Hierbei wird für jedes Testlistenelement τ, für das gilt $c_\tau \neq 0$, ein hierarchisches Ereignisstromelement θ erstellt:

$$\theta = (\infty, a_\tau, c_\tau, \infty).$$

Zusätzlich wird für jedes Testlistenelemente τ_i mit $G_{\tau_i} \neq 0$ ebenfalls ein hierarchisches Ereignisstromelement θ erstellt:

$$\theta = (\infty, a_{\tau_i}, l, g),$$

wobei g der Gradient ist, der sich bei Auswertung der Testliste bei einer Intervalllänge von a_{τ_i} ergibt. Die Beschränkung l des Ereignisstromelements ergibt sich durch die Kosten, die der Gradient g bis zum nächsten Testlistenelement τ_{i+1} verursacht, und wird berechnet durch $l = (a_{\tau_{i+1}} - a_{\tau_i}) \cdot g$. Die so erhaltenen Ereignisstromelemente werden zu einem Ereignisstrom Θ_{AP} zusammengefasst.
Das gleiche wird dann auch für den periodischen Teil durchgeführt. Diesmal wird jedoch vom Offset der Ereignisstromelemente der Wert der Grenze zwischen periodischen und aperiodischem Bereich abgezogen. Die hierbei erhaltenen Elemente werden im Ereignisstrom Θ_P zusammengefasst. Dieser Ereignisstrom wird schließlich als Ereignismuster des periodischen Elements

$$\theta_P = (T, b, n, \Theta_P)$$

dem Ereignisstrom Θ_{AP} hinzugefügt. Dabei ist T die Periode des periodischen Bereichs; b ist die Grenze zwischen periodischem und aperiodischen Bereich und

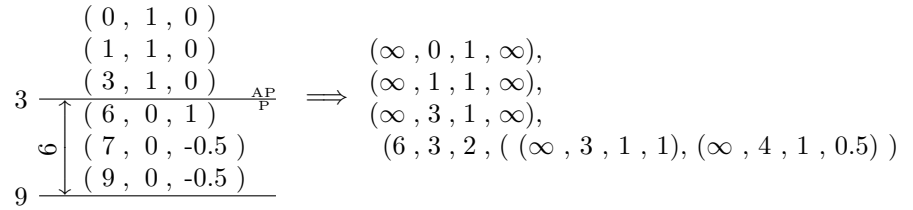


Abbildung 4.5: Generierung eines Ereignisstroms aus einer periodischen Testliste

n sind die Gesamtkosten, die der periodische Bereich innerhalb einer Periode erzeugt.
Auf diese Weise lässt sich ein Ereignisstrom durch einmaliges Traversieren einer Testliste aus dieser erstellen. Dies ist hier einmal beispielhaft in Abbildung 4.5 dargestellt.

4.5 Umsetzung der Operationen

Um die Ereignisströme nun tatsächlich mit dem Realtime Calculus zu verknüpfen, müssen die in den Gleichungen 3.2, 3.3, 3.4 und 3.5 beschriebenen Operationen für die Testlisten umgesetzt werden. Hier folgt nun eine informelle Beschreibung der Vorgehensweise der Algorithmen, die in [2] vorgestellt wurden. Im nächsten Kapitel werden diese dann anhand der hier vorgenommenen Implementierung für die Komplexitätsanalyse näher betrachtet. Dafür wird jedoch vorausgesetzt, dass die zugrunde liegenden Ideen und Arbeitsweisen bekannt sind.

4.5.1 Berechnung der Servicekurven

Die Definition zur Berechnung der oberen ausgehenden Servicekurve ist beinahe identisch mit derjenigen zur Berechnung der unteren ausgehenden Servicekurve.

$$\beta^{l'}(\Delta) = \sup_{0 \leq t \leq \Delta} \{\beta^{l}(t) - \alpha^{u}(t)\} \tag{4.1}$$

$$\beta^{u'}(\Delta) = \sup_{0 \leq t \leq \Delta} \{\beta^{u}(t) - \alpha^{l}(t)\} \tag{4.2}$$

Tatsächlich unterscheiden sie sich nur dadurch, dass einmal die obere Ankunftskurve von der unteren Servicekurve abgezogen wird und das andere Mal die untere Ankunftskurve von der oberen Servicekurve. Da bei der Umsetzung aber kein Gebrauch von den speziellen Eigenschaften des jeweiligen Kurventypes gemacht wird, erfolgt die Berechnung der oberen und unteren, ausgehenden Servicekurve identisch.
Es müssen jeweils zwei Operationen nacheinander ausgeführt werden. Zuerst erfolgt die Subtraktion und anschließend wird auf dem resultierendem Ereignisstrom die Supremumfunktion mit den hier gegebenen Grenzen ausgeführt.

4.5.1.1 Subtraktion

$$\Theta_{Differenz} = \Theta_{Minuend} - \Theta_{Subtrahend}$$

Die Subtraktion zweier Ereignisströme ist insbesondere für approximierte Testlisten relativ einfach zu bewerkstelligen. Hierzu werden zuerst alle Testlisten-Elemente des Subtrahenden negiert. Die Negation eines Elementes geschieht dabei wie folgt:

$$\tau = (a, c, G) \qquad \Rightarrow \qquad -\tau = (a, -c, -G)$$

Durch dieses Vorgehen wird der Funktionsgraph der Testliste an der X-Achse gespiegelt.
Anschließend werden die Elemente beider Testlisten zu einer vereint. Enthalten

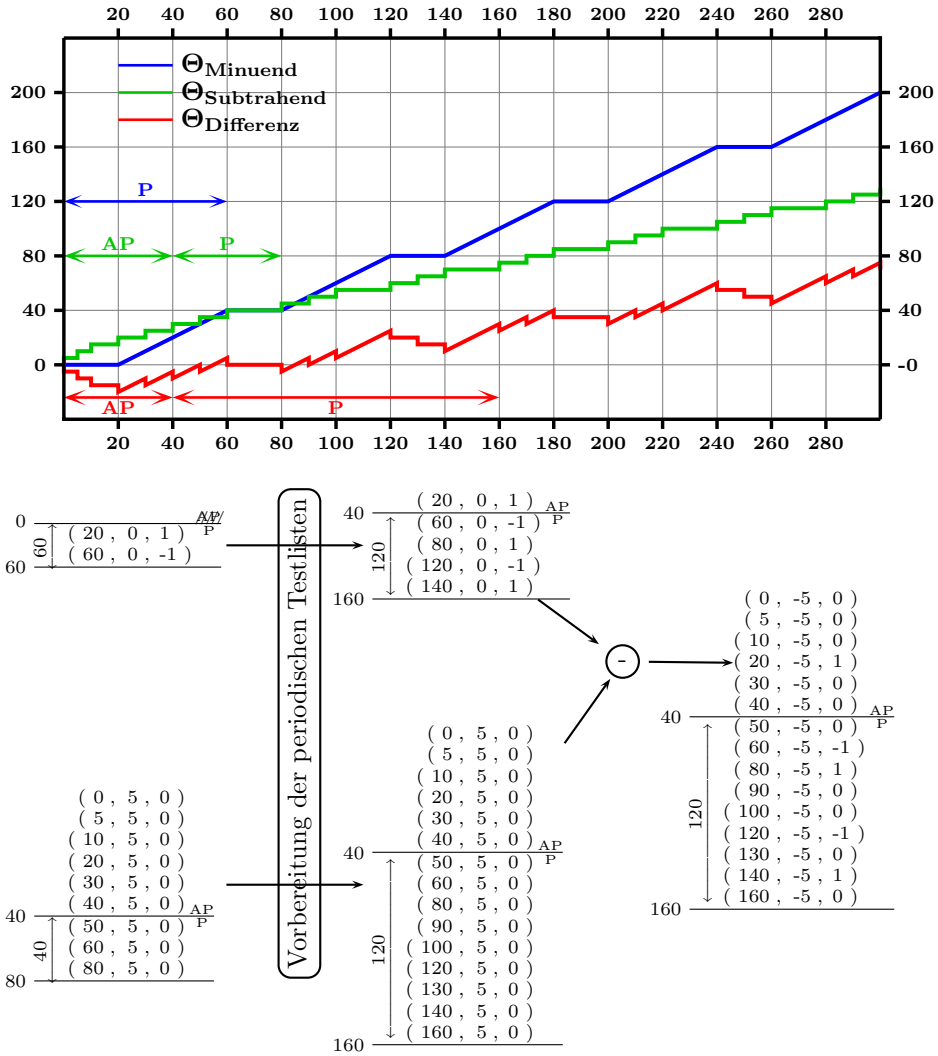


Abbildung 4.6: Subtraktion zweier Ereignisströme

die Testlisten Elemente mit identischem Offset, so werden diese zu einem Element zusammengefasst, dessen Kosten und Gradienten sich aus der Summe der Kosten bzw. Gradienten der ursprünglichen Elemente bilden.
Bei Testlisten mit periodischen Bereichen kann sich der Aufwand jedoch stark vergrößern. Dies wird leicht an einem Beispiel deutlich, dass in Abbildung 4.6 zu sehen ist.
Hier wird von einer rein periodischen Servicekurve (Minuend) eine Ankunftskurve (Subtrahend) abgezogen, die sowohl einen periodischen als auch einen aperiodischen Anteil besitzt. Dazu wird zunächst einmal die maximale Länge der resultierenden Testliste ermittelt. Die Länge des aperiodischen Bereichs von $\Theta_{Differenz}$ ergibt sich dabei aus dem Maximum der aperiodischen Bereichslängen von $\Theta_{Minuend}$ und $\Theta_{Subtrahend}$. Dies entspricht hier der aperiodischen Bereichslänge des Subtrahenden. Die Länge des periodischen Bereichs ermittelt sich dagegen aus dem kleinsten, gemeinsamen Vielfachen der periodischen Bereichslängen von $\Theta_{Minuend}$ und $\Theta_{Subtrahend}$.
Nun werden die Testlisten von $\Theta_{Minuend}$ und $\Theta_{Subtrahend}$ durch wiederholtes Anhängen des periodischen Bereichs auf die ermittelte Länge erweitert und auf die gleiche Weise wie approximierte Testlisten von einander abgezogen.
Jetzt zeigt sich auch das große Problem bei der Berechnung von periodischen Ereignisströmen. Durch die Bildung des kleinsten, gemeinsame Vielfachen können die Testlisten sehr schnell einen sehr großen periodischen Bereich erhalten. Im Beispiel aus Abbildung 4.6 tritt dies noch nicht sehr deutlich hervor, da die Perioden von $\Theta_{Minuend}$ und $\Theta_{Subtrahend}$ einen relativ großen gemeinsamen Teiler besitzen. Dieses Problem war jedoch zu erwarten, da es generell bei der Analyse von periodischen Tasksystemen auftritt.

4.5.1.2 SupSimple

$$\Theta_{Service\_out} = \sup_{0 \leq t \leq \Delta} \{\Theta_{Differenz}\}$$

Die durch die Subtraktion erhaltene Testliste entspricht nicht den Anforderungen einer Servicekurve. Insbesondere ist die Monotonie-Eigenschaft verletzt. Die Supremumfunktion korrigiert dies, indem sie Kurvenbereiche, die unterhalb eines lokalen Maximums, das vor diesem Bereich auftritt, auf eben dieses Maximum anhebt. Den Effekt der Supremum-Operation kann man in Abbildung 4.7 erkennen. Dort wird das Supremum über dem Ereignisstrom gebildet, der sich als Differenz aus dem Beispiel im vorherigen Abschnitt (4.5.1.1) ergab.
Albers et al. stellten in [2] einen Algorithmus für die hier betrachtete Supremumfunktion vor. Da für die obere Ankunftskurve auch eine Supremumfunktion benötigt wird, die jedoch wesentlich aufwendiger zu berechnen ist, wird zur Unterscheidung die Funktion zur Berechnung der hier benötigten Supremumoperation als *SupSimple* bezeichnet.
Der Algorithmus arbeitet ebenfalls auf Testlisten, die keine Informationen über das periodische Verhalten des Ereignisstroms beinhalten. Die Idee des Algorithmus ist, die Testliste einmal zu durchlaufen und dabei eine neue Testliste zu generieren, die das Ergebnis beinhaltet. Dabei wird das Maximum des bereits bearbeiteten Testlistenabschnitts immer mitgeführt. In die resultierende Testliste werden dann nur die Elemente eingefügt, die in den Bereichen oberhalb des aktuellen Maximums liegen. Gegebenenfalls müssen an den Grenzen dieser

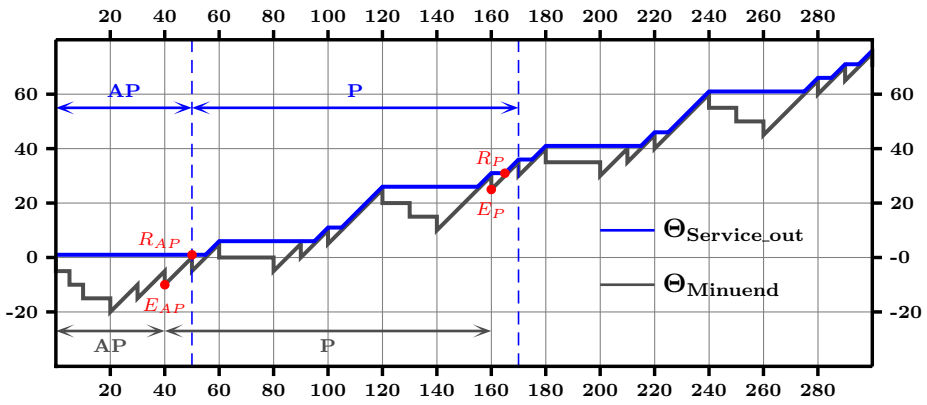


Abbildung 4.7: SupSimple angewendet auf der Differenz zweier Ereignisströme

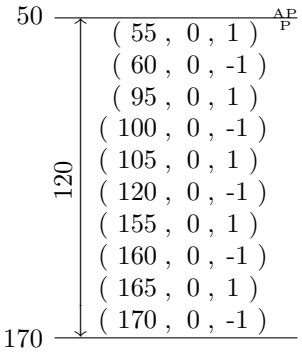


Abbildung 4.8: Ergebnis der SupSimple-Operation aus Abbildung 4.7

Bereiche noch zusätzliche Elemente generiert werden.
Bei Testlisten, die zusätzliche Informationen zum periodischem Verhalten beinhalten, kann dieses Verfahren auch benutzt werden. Allerdings muss hierbei ein vorbereitender Rechenschritt durchgeführt werden. Für $\Theta_{Service\_out}$ müssen dabei die Informationen zum periodischen Bereich ermittelt werden. Es ist leicht ersichtlich, dass die Periode von $\Theta_{Service\_out}$ identisch mit der von $\Theta_{Differenz}$ ist. Nur in einem Fall gibt es davon eine Ausnahme und zwar wenn die durchschnittliche Steigung des periodischen Bereichs von $\Theta_{Differenz}$ nicht positiv ist. In diesem Fall gibt es ein lokales Maximum, über das die Kurve nicht mehr hinaus steigt. Ab diesem Maximum hat die Kurve von $\Theta_{Service\_out}$ eine Steigung von 0 und besitzt somit keinen periodischen Anteil mehr.
Nun wird noch die Länge des aperiodischen Bereichs benötigt. Wenn die durchschnittliche Steigung des periodischen Bereichs von $\Theta_{Differenz}$ nicht positiv ist, so besitzt $\Theta_{Service\_out}$ -wie oben erwähnt- keinen periodischen Anteil mehr und die Länge des aperiodischen Bereichs kann auf den Offset des letzten Testlistenelements von $\Theta_{Service\_out}$ gesetzt werden. In den anderen Fällen kann sich die Länge von derjenigen von $\Theta_{Differenz}$ unterscheiden. Das passiert genau dann, wenn es entweder innerhalb des aperiodischen Bereichs einen Wert gibt, der größer ist, als derjenige am Ende des aperiodischen Bereichs, oder wenn es innerhalb des periodischen Bereichs einen Wert gibt, der größer ist, als derjenige am Ende des periodischen Bereichs. Beide Fälle haben die gleiche Wirkung. In Abbildung 4.7 kann man diesen Effekt erkennen. Dort ist der Wert am Ende des aperiodischen Bereichs mit E_{AP} und der Wert am Ende des periodischen Be-

reichs mit E_P gekennzeichnet. Relevant ist jedoch immer nur der Punkt, dessen Funktionswert am weitesten vom entsprechenden Funktionswert von $\Theta_{Service\_out}$ entfernt ist. In diesem Fall ist das E_{AP}. Da der Wert unterhalb von $\Theta_{Service\_out}$ liegt, wird der Anfang des periodischen Bereichs von $\Theta_{Differenz}$ nicht für die Testliste von $\Theta_{Service\_out}$ verwendet. Da der aperiodische Bereich aber nur vor der ersten Periode auftaucht, wird der Anfang des periodischen Bereichs für die weiteren Perioden jedoch benötigt. Somit unterscheidet sich dieser Bereich nur in der ersten Periode und muss daher zum aperiodischen Bereich gezählt werden. Die Länge des Bereichs erstreckt sich vom Anfang des periodischen Bereichs bis zu dem Punkt, an dem $\Theta_{Differenz}$ wieder den Wert von $\Theta_{Service\_out}$ erreicht, da ab hier der Verlauf von $\Theta_{Differenz}$ wieder relevant ist für den Verlauf von $\Theta_{Service\_out}$. Dieser Punkt ist in Abbildung 4.7 mit R_{AP} gekennzeichnet.
Hätten man in diesem Beispiel den Fall gehabt, dass E_P der relevante Punkt ist, so wäre der Effekt der gleiche, jedoch mit einer anderen Begründung. In diesem Fall würde der Anfang jeder Periode von $\Theta_{Differenz}$ nicht für den Verlauf von $\Theta_{Service\_out}$ genutzt werden, da es immer in der vorherigen Periode einen Maximalwert gibt, der dafür sorgt, dass der Anfang des periodischen Bereichs von $\Theta_{Differenz}$ immer unterhalb von $\Theta_{Service\_out}$ liegt. Da es allerdings zur ersten Periode keine vorherige gibt, ist der Anfang der ersten Periode von $\Theta_{Differenz}$ relevant für den Verlauf von $\Theta_{Service\_out}$. Da dies auch hier nur für die erste Periode gilt, muss auch in diesem Fall die Länge des aperiodischen Bereichs von $\Theta_{Service\_out}$ um die Länge des Bereichs von E_P bis R_P erweitert werden.
Das bedeutet, dass als Vorverarbeitungsschritt für die Supremumoperation der maximale Abstand vom Funktionswert von E_P und E_{AP} zum größten vorherigen Maximum ermittelt werden muss. Nun kann berechnet werden, wie lange der periodische Bereich von $\Theta_{Differenz}$ benötigt, um diesen Abstand als Funktionswert zu erreichen. Die Testliste von $\Theta_{Differenz}$ muss dann um genau diese Länge mittels ihres periodischen Bereichs erweitert werden.
Damit ist die Vorverarbeitung abgeschlossen und die SupSimple-Funktion kann wie für aperiodische Testlisten auf der verlängerten Testliste von $\Theta_{Differenz}$ durchgeführt werden.

4.5.2 Berechnung der unteren Ankunftskurve

$$\alpha^{l'}(\Delta) = \inf_{0 \leq t \leq \Delta} \{\alpha^l(\Delta - t) + \beta^l(t)\}$$

Wie bereits in Kapitel 3 beschrieben, erfolgt die Berechnung der unteren Ankunftskurve, indem für jedes Intervall die untere Ankunfts- und Servicekurve so auf das Intervall aufgeteilt wird, dass sich der kleinste Wert ergibt. Daher trägt die Funktion, die dies berechnet, den Namen *MinimumSplit*. Diese Operation wird sowohl für die untere als auch für die obere Ankunftskurve verwendet. Die Vorgehensweise, die im Folgenden vorgestellt wird, ist daher auch als Teilfunktion für die Berechnung der oberen Ankunftskurve verwendbar.
Das Vorgehen zur Berechnung dieser Operation wird zuerst anhand einfacher Testlisten betrachtet, die keine periodischen Bereiche besitzen.
Für solche Testlisten stellten *Albers* und auch den MinimumSplit-Algorithmus vor, der dies bewerkstelligt. Die Arbeitsweise dieses Algorithmus soll hier nun erläutert werden. Wie bereits in Abschnitt 3.2.2.3 erwähnt, teilt diese Infimum-Operation die aktuell betrachtete Intervalllänge so auf die beiden beteiligten

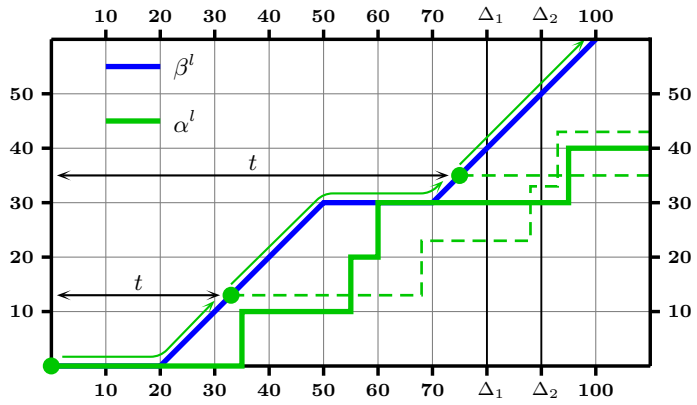


Abbildung 4.9: Illustration der MinimumSplit-Operation

Kurven auf, dass als Summe $(\alpha^l(\Delta - t) + \beta^l(t))$ der kleinst mögliche Wert erzielt wird. Da Δ in $\mathbb{R}^+$ definiert ist, ist es nicht machbar, für jedes mögliche Δ jeweils alle möglichen Werte für t zu wählen und die Summe zu bilden. Nun kann man sich anhand einer grafischen Darstellung anschauen, wie der Werte für ein einzelnes, festes Δ bestimmt werden kann.
In Abbildung 4.9 soll das beispielhaft für $\Delta_1 = 80$ und $\Delta_2 = 90$ gezeigt werden. Um nun $\alpha^{l'}(\Delta_1)$ zu berechnen muss $\alpha^l(\Delta_1 - t) + \beta^l(t)$ für alle $0 \leq t \leq \Delta_1$ berechnet werden. Das kleinste Ergebnis ist dann der Funktionswert von $\alpha^{l'}(\Delta_1)$. Für zwei Werte von t ist die Berechnung grafisch in der Abbildung dargestellt. Dabei wird der Ursprung von α^l entlang des Funktionsverlaufs von β^l bis zur Position t verschoben. Hierbei wird der Punkt $(\Delta_1 - t, \alpha^l(\Delta_1 - t))$ auch um t in X-Richtung und $\beta^l(t)$ in Y-Richtung verschoben, und erreicht daher den Punkt $(t, \beta^l(t) + \alpha^l(\Delta_1 - t))$. Der Funktionswert der verschobenen Funtkion ist an dieser Stelle also der für das aktuelle t gesuchte Wert.
Betrachtet man das gleiche für Δ_2, so ist das Vorgehen identisch. Auch hier muss α^l entlang des Verlaufs von β^l verschoben werden. Es liegt also nahe, dass diese Berechnungen nicht für jedes Δ erneut durchgeführt werden müssen. Es reicht aus, wenn α^l für jedes $t > 0$ nur einmal verschoben wird und dann das Minimum aus allen verschobenen Funktionen gebildet wird.
Nun gibt es theoretisch unendlich viele Werte für t. Da hier jedoch spezielle Funktionen verwendet werden, die ausschließlich aus endlich vielen linearen Segmenten zusammengesetzt sind, kann die Anzahl der Verschiebungen begrenzt werden.
Die Idee hierbei ist, α^l nur an die Punkte zu verschieben, bei denen sich die Steigung von β^l ändert. Da eine solche Änderung immer durch ein Testlisten-Element dargestellt wird, ist die Anzahl der Verschiebungen durch die Länge der Testliste von β^l begrenzt.
Jetzt stellt sich nur noch die Frage, ob das ausreichend ist, um ein korrektes Ergebnis zu erhalten. Leider ist es das nicht. In Abschnitt 3.2.2.3 wurde bereits erläutert, dass der Term $\alpha^l(t) + \beta^l(\Delta - t)$ innerhalb der Infimum-Operation austauschbar ist mit $\alpha^l(\Delta - t) + \beta^l(t)$. Daher müsste es egal sein, ob nun α^lentlang von β^l verschoben wird, oder ob dies umgekehrt geschieht.

$$\alpha^l = \begin{matrix}(\ 20\ ,\ 0\ ,\ 0.25\ )\\ (\ 60\ ,\ 0\ ,\ 0.75\ )\\ (\ 80\ ,\ 0\ ,\ -1\ )\\ (\ 100\ ,\ 0\ ,\ 0.25\ )\\ (\ 140\ ,\ 0\ ,\ 0.1\ )\end{matrix} \qquad \beta^l = \begin{matrix}(\ 40\ ,\ 0\ ,\ 0.5\ )\\ (\ 80\ ,\ 0\ ,\ -0.5\ )\\ (\ 120\ ,\ 0\ ,\ 0.5\ )\\ (\ 160\ ,\ 0\ ,\ -0.5\ )\\ (\ 200\ ,\ 0\ ,\ 0.25\ )\end{matrix}$$

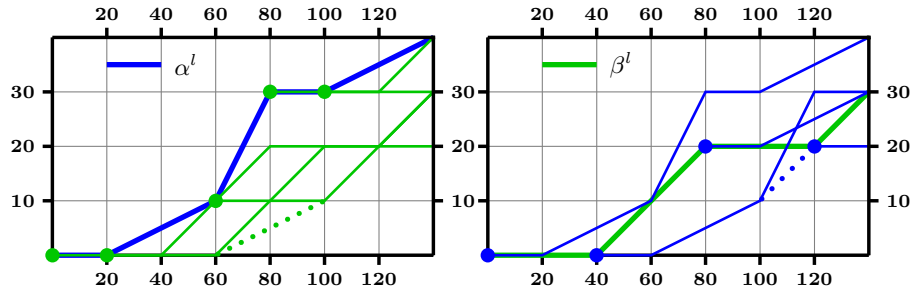


Abbildung 4.10: Verschieben entlang der Testlisten-Elemente

Für beide Fälle ist dies in Abbildung 4.10 dargestellt. Links ist dort eine untere Ankunftskurve zu sehen. Die Punkte, an denen sich die Steigung von α^l ändert sind markiert. Außerdem sind die Kurvenverläufe von β^l eingezeichnet, die jeweils an die markierten Punkte verschoben wurden. Das ergibt ein unübersichtliches Gewirr von Liniensegmente. Nach den obigen Überlegungen sollte das Minimum all dieser Verläufe nun das Ergebnis der MinimumSplit-Operation sein. Außerdem sollte man das gleiche Ergebnis erhalten, wenn man anstelle der Servicekurve die Ankunftskurve verschiebt. Dies ist im rechten Koordinatensystem dargestellt.
Hier ist nun zu erkennen, dass sich die minimalen Verläufe unterscheiden. Die gestrichelten Linien geben dabei die Segmente an, die in der jeweiligen Grafik fehlen. Um dieses Problem zu lösen, kann man nun einfach sowohl α^l als auch β^l verschieben und das Minimum über alle Kurvenverläufe bilden, die dabei entstehen.
Möchte man die Infimum-Operation nun mit Testlisten durchführen, die zusätzlich Informationen zum periodischen Verhalten beinhalten, so ist das prinzipiell auch bei dieser Operation möglich. Wie auch bei der Subtraktion und der SupSimple-Operation müssen hierfür zunächst einige Werte ermittelt werden.
Wie das Vorgehen dazu aussehen könnte, soll zuerst anhand von rein periodischen Testlisten gezeigt werden, die keinen aperiodischen Bereich besitzen. An einem Beispiel soll nun demonstriert werden, wie sich die Berechnungen bei periodischen Funktionen verhalten. Über die periodische Information kann man im Prinzip unendlich viele Testlisten-Elemente erzeugen, an die die jeweils andere Liste verschoben werden soll. Nach dem kleinsten gemeinsamen Vielfachen (kgV) der Perioden wiederholen sich die Muster jedoch. An einem sehr einfachen Beispiel in Abbildung 4.11 erkennt man dies sehr gut. Im linken Koordinatensystem sind zwei verschobene Graphen von β^l eingezeichnet. Um die verschobenen Graphen unterscheiden zu können, soll im Folgenden $\upsilon_i(\beta^l)$, den Graphen von β^l bezeichnen, dessen Ursprung an das i-te Element der Testliste von α^u verschoben wurde. Entsprechend soll auch $\upsilon_i(\alpha^l)$ definiert sein.
Der Graph von $\upsilon_1(\beta^l)$ wurde nun an die Position (30,0) und der von $\upsilon_3(\beta^l)$ eine

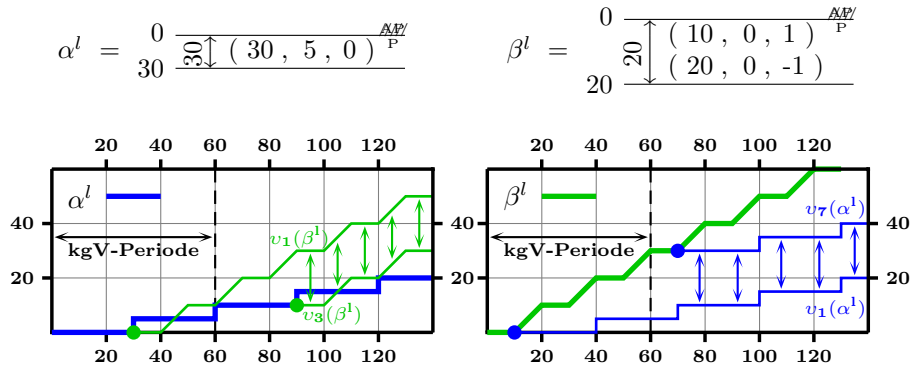


Abbildung 4.11: Parallelität beim Verschieben von Testlisten

kgV-Periode später an die Position (90,10) verschoben. Es ist nun leicht zu sehen, dass $\upsilon_3(\beta^l)$ parallel $\upsilon_1(\beta^l)$ verläuft. Der gleiche Effekt ist beim Verschieben von α^l im rechten Koordinatensystem bei den Graphen von $\upsilon_7(\alpha^l)$ und $\upsilon_1(\alpha^l)$ zu erkennen.
Allerdings liegt $\upsilon_3(\beta^l)$ unterhalb von $\upsilon_1(\beta^l)$ während im rechten Koordinatensystem der Graph von $\upsilon_7(\alpha^l)$ oberhalb von $\upsilon_1(\alpha^l)$ liegt. Das ist darauf zurückzuführen, dass die durchschnittliche Steigung von α^l kleiner ist als die von β^l. Deswegen sind in diesem Beispiel die Verläufe von α^l, die hinter die kgV-Periode geschoben werden, nicht mehr relevant für das Ergebnis. Es reicht daher aus, wenn nach der kgV-Periode nur noch β^l verschoben wird. Damit einher geht der Effekt, dass sich die Periodizität des Ergebnissses nach der kgV-Periode nur nach der Kurve mit der geringsten Steigung richtet. Das ist in diesem Fall α^l.
Etwas stärker auf den Realtime Calculus bezogen bedeutet dies, dass die Periodizität der ausgehenden Ankunftskurve eines Task gleich der Periodizität der eingehenden Ankunftskurve ist, wenn der Task planbar ist. Denn wenn er dies ist, muss α^l eine geringere durchschnittliche Steigung haben als β^l. Ansonsten würde auf lange Sicht mehr Prozessorzeit angefordert, als geliefert werden kann. Die Periode des Ergebnisses kann also recht einfach ermittelt werden. Wählt man für die Länge des aperiodischen Bereichs die kgV-Periode, so ist man auch hier auf der sicheren Seite, auch wenn in vielen Fällen ein kleinerer aperiodischer Bereich gefunden werden kann. Das soll an dieser Stelle jedoch nicht näher ausgeführt werden.
Bisher wurden nur Testlisten betrachtet, die entweder nur einen periodischen oder nur einen aperiodischen Bereich besitzen. Eine unerwartete Problematik zeigt sich jedoch bei Testlisten,die beides besitzen. In Abbildung 4.12 ist hierzu ein Beispiel dargestellt. Es zeigt eine untere Ankunftskurve mit einem ungewöhnlichen aperiodischen Bereich, der das Problem in einem Extremfall beschreibt. Ein solcher Fall kann eintreten, wenn der Beginn des aperiodischen Bereichs eine relativ geringe durchschnittliche Steigung hat, das Ende aber eine sehr starke Steigung. Die Schwierigkeit besteht nun darin, dass das Ergebnis einen sehr langen aperiodischen Bereich erhalten kann.
In der Beispielgrafik ist die verschobene Kurve $\upsilon_1(\beta^l)$ eingezeichnet. Ihr Startpunkt liegt im aperiodischen Bereich und hat im Vergleich zu den folgenden

Startpunkten von $\upsilon_1(\beta^l), \upsilon_2(\beta^l), \cdots$ usw. nur einen sehr geringen Funktionswert. Daher tragen diese Kurven nicht zum Verlauf des Ergebnisses bei. Dies ändert sich erst wieder in dem Bereich, in dem $\upsilon_1(\beta^l)$ den Graphen von α^l schneidet. Da die Steigung von α^l jedoch nur geringfügig kleiner ist, als die von $\upsilon_1(\beta^l)$ liegt dieser Bereich sehr weit entfernt. Da aber erst hier der eigentliche periodische Bereich beginnt, hat das Ergebnis in diesem Fall einen sehr großen aperiodischen Bereich.

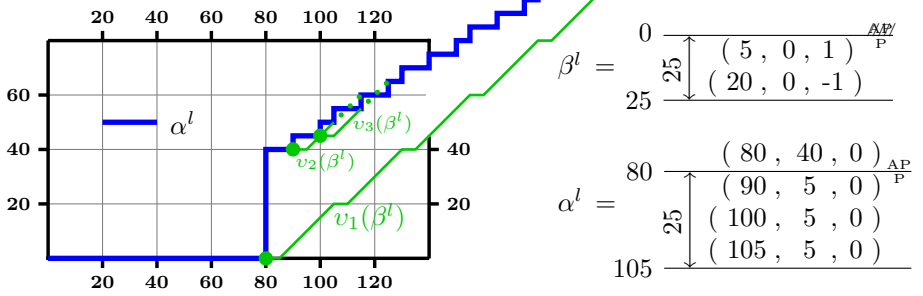


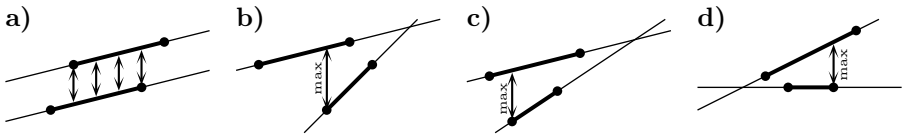
Abbildung 4.12: Beispiel problematischer Testlisten für die MinimumSplit-Operation

Nun kann man sich fragen, was daran so gravierend ist, da schließlich auch die Bildung des kleinsten gemeinsamen Vielfachen der Perioden zu sehr langen Testlisten führt. Für das kgV kann jedoch vor der Analyse immer eine obere Grenze bestimmt werden, da die Perioden aller beteiligter Kurven bekannt sind. Die Obergrenze für die Länge des aperiodischen Bereichs bei der MinimumSplit-Operation kann ohne weitere Überlegungen jedoch nicht so einfach bestimmt werden.
Im obigen Beispiel würde sich der Schnittpunkt von α^l und $\upsilon_1(\beta^l)$ immer weiter in X-Richtung entfernen, je mehr sich die durchschnittlichen Steigungen der beiden Kurven ähneln.
Das Problem erscheint lösbar. Jedoch sind hierzu weitere Anstrengungen nötig. An dieser Stelle wäre es sicherlich sinnvoll, sich erst einmal Gedanken darüber zu machen, wodurch solche Situationen entstehen können, und mit welchen Annahmen sie sich begrenzen lassen. Dann muss eine Strategie gefunden werden, die diese Situationen effizient löst. Die hier benutzten zweigeteilten Testlisten scheinen dafür jedenfalls nicht besonders geeignet zu sein.
Betrachtet man noch einmal die Kurvenverläufe in Abbildung 4.12, so erkennt man leicht, dass sich das Ergebnis der MinimumSplit-Operation im Bereich von 40 bis zum Schnitt der dargestellten Verläufe bei etwa 250 mit der Periodizität von β^l verhalten wird. Danach wird es sich jedoch mit der Periodizität von α^l verhalten, da α^l die geringere durchschnittliche Steigung hat. Die zweigeteilten Testlisten können aber nur eine Periodenlänge handhaben. Da das Ereignisstrommodell Elemente mit verschiedenen Perioden beinhalten kann, könnte es an dieser Stelle sinnvoll sein, die Umsetzung einer exakten Systemanalyse auf dem Ereignisstrommodell selbst durchzuführen. Die Idee bei den zweigeteilten Testlisten bestand jedoch darin, die Algorithmen aus [2] mit möglichst geringem Mehraufwand in der Implementierung auch für eine exakte Analyse zu verwen-

den.
Da der zusätzliche Aufwand hier sehr groß erscheint, soll die exakte Analyse an dieser Stelle nicht weiter verfolgt werden. Im weiteren Verlauf der Arbeit wird daher der Fokus auf der Komplexitätsanalyse der bestehenden Algorithmen liegen.

4.5.3 Berechnung der oberen Ankunftskurve

Im vorherigen Abschnitt wurde erläutert, wie die Infimum-Operation zur Berechnung der unteren Ankunftskurve für Testlisten mit der MinimumSplit-Funktion realisiert werden kann. Eine Infimum-Operation mit gleichen Grenzen und gleichem Aufbau wird auch zur Berechnung der oberen Ankunftskurve verwendet. Daher kann auch hier die MinimumSplit-Funktion benutzt werden.
Der Unterschied zur unteren Ankunftskurve besteht darin, dass die Parametermenge der Infimumoperation hier durch die Funktion $\sup_{v\geq 0}\{\alpha^u(t+v) - \beta^l(v)\} + \beta^u(\Delta - t)$ gebildet wird. Die dabei verwendete Supremum-Operation unterscheidet sich von der SupSimple-Operation, die zur Berechnung der Servicekurven verwendet wird. Daher wird an dieser Stelle eine weitere Funktion verwendet, die die Berechnung dieser Operation für Testlisten durchführt. Sie wird benötigt, um die maximale Menge an angeforderter aber noch nicht bearbeiteter Rechenzeit zu beschreiben. Um diese Supremum-Operation von der SupSimple-Operation zu unterscheiden, wird sie im weiteren als *MaximumRequest* bezeichnet.
Um die Arbeitsweise dieser Funktion zu beschreiben, sollte ein näherer Blick auf den Term $\sup_{v\geq 0}\{\alpha^u(t+v) - \beta^l(v)\}$ geworfen werden. Setzt man $t = 0$, so ist leicht zu erkennen, dass dann die größte Differenz zwischen α^u und β^l berechnet wird. Durch den Parameter t kann nun aber α^u entlang der X-Achse verschoben werden. Gesucht ist an dieser Stelle also eine Funktion $s(t)$, die die größte Differenz bei einer Verschiebung von α^u um t angibt.
Die Berechnung für $t = 0$ durchzuführen ist sehr einfach. Es muss hierzu über beide Listen iteriert werden. Bei jedem Testlisten-Element wird dann die aktuelle Differenz zwischen α^u und β^l errechnet. Die größte Differenz ist dann der für $t = 0$ gesuchte Wert. Die Differenz zwischen den beiden Kurven muss dabei immer nur für die Offsets aller Testlisten-Elemente gebildet werden. An Stellen zwischen zwei Testlisten-Elementen ist das nicht nötig. Warum das so ist, ist hier kurz dargestellt.
Da die Testlisten Funktionen aus linearen Segmenten darstellen, muss für jedes v immer nur die Differenz zwischen zwei Linien-Segmenten betrachtet werden. Hier sind einige Beispiele abgebildet.



Sind die Segmente parallel, so ist offensichtlich, dass die Differenz überall gleich ist, und der maximale Wert daher auch an den Endpunkten der Segmente ermittelt werden kann, die durch die Testlisten-Elemente beschrieben werden (s. Beispiel a). Betrachtet man jedoch in allen anderen Fällen die Geraden, auf denen die Segmente liegen, so werden sie sich irgendwo schneiden. Je weiter man

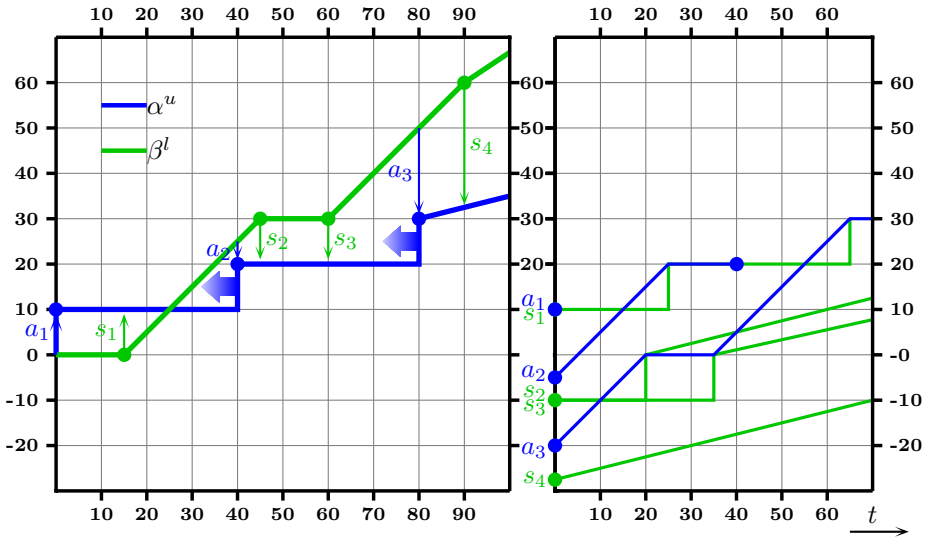


Abbildung 4.13: Berechnung der Supremum-Funktion

sich nun vom Schnittpunkt entfernt, desto größer wird die Differenz zwischen den Geraden. Für die Segmente wird also die größte Differenz an dem Punkt erreicht, der am weitesten vom Schnittpunkt entfernt liegt. Das muss an einem Endpunkt eines Segmentes sein und wird daher durch ein Testlisten-Element repräsentiert. Dementsprechend muss die Berechnung für $t = 0$ nur für die Endpunkte aller Segmente durchgeführt werden. Da diese durch die Elemente der Testlisten beschrieben werden, kann die Berechnung sehr leicht auf den Testlisten durchgeführt werden.
Verschiebt man nun α^u durch ein beliebiges t und stellt die Berechnung erneut an, so muss natürlich auch hier wieder nur die Differenz an den Testlisten-Elementen ermittelt werden. Indem man dies für alle $t > 0$ macht, berechnet man die Supremumoperation.
In Abbildung 4.13 sind links zwei Beispielkurven für α^u und β^l zu sehen. Für $t = 0$ sind die Differenzen für alle Testlisten-Elemente von α^u zum Verlauf von β^l eingezeichnet und mit a_1 bis a_3 beschriftet. Ebenso sind die Differenzen aller Testlisten-Elemente von β^l zum Verlauf von α^u eingetragen und mit s_1 bis s_4 beschriftet.
Diese Differenzen sind nun rechts in der Abbildung bei $t = 0$ eingetragen. Verschiebt man nun α^u und betrachtet, wie sich die Differenzen mit steigendem t verändern, so ergibt sich für jedes Testlisten-Element aus beiden Listen eine Funktion die von t abhängt. Eine solche Funktion wird im folgenden als Differenzfunktion bezeichnet. Die Graphen aller Differenzfunktionen sind für das Beispiel rechts in der Abbildung eingezeichnet.
Für jedes t ist nun der maximale Wert aller Graphen gesucht. Daher kann die Berechnung der Supremum-Funktion durchgeführt werden, indem das Maximum über alle Differenzfunktionen gebildet wird.
Der Verlauf einer solchen Funktion lässt sich einfach herleiten. Betrachtet man einen beliebigen Punkt auf dem Graphen von β^l und verschiebt dann α^u, so erkennt man sehr leicht, dass sich die Differenz zwischen den Kurven an dieser

Stelle genau wie der Verlauf von α^u verhält. Daher sind die Verläufe der Differenzfunktionen, die in der rechten Abbildung an den Punkten s_1 bis s_4 beginnen, beinahe identisch mit dem Verlauf von α^u. Sie sind lediglich um entsprechende Werte in X- und Y-Richtung verschoben.
Betrachtet man dagegen einen beliebigen Punkt auf dem Graphen von α^u und verschiebt α^u in X-Richtung, so sieht man hier, dass sich die Differenz an dieser Stelle, nach dem Verlauf von β^l richtet. Da α^u jedoch mit steigendem t nach links verschoben wird, entwickelt sich die Differenzfunktion hier nicht mit dem Verlauf von β^l, sondern genau entgegengesetzt. Daher entsprechen die Graphen die rechts in der Abbildung an den Punkten a_1 bis a_3 beginnen, der Punktspiegelung von β^l am Ursprung. Zudem sind auch diese Funktionen in X- und Y-Richtung verschoben.
Für die spätere Komplexitätsuntersuchung ist an dieser Stelle noch eine weitere Eigenschaft interessant. Es sollen nun für je ein Testlisten-Element τ_α aus α^u und eines τ_β aus β^l die Punkte auf den jeweiligen Graphen betrachtete werden. Dabei muss der Offset von τ_α größer sein als der von τ_β. Wählt man nun $t = a_{\tau_\alpha} - a_{\tau_\beta}$, dann wird der Punkt von τ_α an die Stelle von τ_β geschoben. Daher werden die Differenzfunktionen für diese Testlisten-Elemente für $t = a_{\tau_\alpha} - a_{\tau_\beta}$ den gleichen Wert haben, und sich somit dort schneiden.
Da die Graphen der Kurven immer einen Knick bei einem Testlisten-Element haben, besitzen auch die Graphen der angesprochen Differenzfunktionen bei $t = a_{\tau_\alpha} - a_{\tau_\beta}$ jeweils einen Knick und benötigen daher für diese Stelle ein Testlisten-Element. Das bedeutet, dass für die Testliste einer beliebigen Differenzfunktion die Offsets aller Testlisten-Elemente mindestens einmal in der Testliste einer anderen Differenzfunktion auftauchen. Diese Eigenschaft wird im nächsten Kapitel verwendet, um eine Obergrenze für die maximale Länge der Testliste von $\alpha^{u'}$ zu ermitteln.

4.5.4 Implementierung

Die hier vorgestellten Algorithmen wurden innerhalb eines Java-Frameworks umgesetzt. Mit ihm ist es möglich, die Komponenten, die für den Realtime Calculus benötigt werden, abzubilden. Dazu können Objekte für Tasks und Verar-

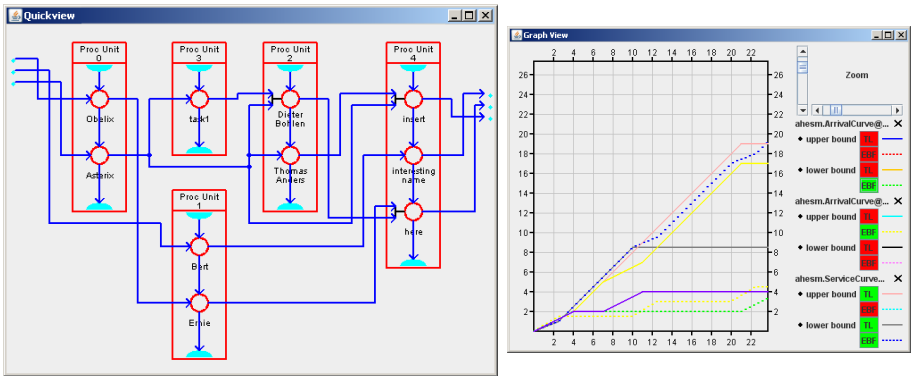


Abbildung 4.14: Grafische Ansicht eines Beispielsystems und Detailansicht einiger Kurven

beitungseinheiten angelegt werden, die wiederum mit Ankunftskurven verknüpft werden können. Die Verknüpfung der Servicekurven ist sowohl manuell, als auch automatisch durch die Wahl eines Schedulers möglich. Bisher ist jedoch nur ein Deadline monotoner Scheduler implementiert.
Aufbauend auf diesem Framework wurde ein weiteres Paket erstellt, dass die Visualisierung eines Systems ermöglicht. Es ist jedoch recht rudimentär und ermöglicht lediglich die Ansicht des Gesamtsystems und eine detailliertere Ansicht der Ankunfts- und Servicekurven.

Kapitel 5

Komplexitätsanalyse

Der Laufzeitaufwand der Analysealgorithmen ist ein wichtiges Kriterium für die Güte dieser Algorithmen. Dieses Kapitel wird sich daher mit der Untersuchung des Aufwands der während dieser Arbeit vorgenommenen Implementierung beschäftigen.
Dazu werden zuerst die notwendigen theoretischen Grundlagen und Notationen der Komplexitätstheorie erläutert, soweit sie für diese Arbeit notwendig sind.
Daran anschließend wird die Komplexität der implementierten Algorithmen untersucht. Dies geschieht schrittweise, beginnend bei einfachen Hilfsfunktionen bis hin zur vollständigen Systemanalyse.

5.1 Theoretische Hintergründe

Die Anfänge der Komplexitätstheorie finden sich schon sehr früh zu Beginn des Computerzeitalters. Die Anreize entstanden dabei gar nicht so sehr aus der theoretischen Sicht der Informatik, sondern waren stark praktisch begründet. Es zeigte sich, dass der Computer bei sehr vielen Problemen des Alltags helfen kann. Jedoch benötigt er dafür Zeit und Speicherplatz. Für große Probleme steigt natürlich der Zeit- und Platzbedarf. Nicht umsonst gibt es seitdem ein andauerndes Rennen nach immer schnelleren Rechnern und größeren Speichern. Dennoch ist auf allen Computern Prozessorleistung und Speicherplatz begrenzt. Es bestand also sehr schnell Bedarf nach Metriken, mit denen die Güte von Algorithmen hinsichtlich dieser Größen gemessen werden kann.
Eine der wichtigsten Aufgaben der Komplexitätstheorie ist dabei die Aufteilung von Problemen in Klassen, die Aufschluss darüber geben, wie effizient ein Problem lösbar ist. Das ist insbesondere bei Problemen wichtig, die nicht effizient lösbar sind. Denn auch wenn es deprimierend ist, zu wissen dass ein Problem nicht effizient lösbar ist, so ist dieses Wissen immer noch bedeutend besser als die vergebliche Suche nach einem gutem Algorithmus.

5.1.1 Komplexitätsmaße

Die Komplexität von Algorithmen wird vor allem an ihrem Zeit- (Time) und Platzbedarf (Space) gemessen. Für Time werden dabei die Rechenschritte ge-

messen, die ein Algorithmus zur Problemlösung benötigt, und für SPACE der benötigte Speicherplatz. In der Praxis wird meistens der Zeitkomplexität mehr Beachtung geschenkt. Dies liegt zum großen Teil daran, dass der Platzbedarf oftmals nicht so kritisch ist. Außerdem ist es häufig wesentlich einfacher einen hinsichtlich SPACE optimierten Algorithmus zu erstellen, als einen auf TIME optimierten.
Da bei einigen Algorithmen dieser Arbeit auch der Platzbedarf eine gewisse Rolle spielt, soll hier jedoch näher darauf eingegangen werden.
Im Unterschied zu TIME unterscheiden sich manche Definitionen von SPACE. Meistens gibt dieses Maß die Anzahl aller Speicherzellen an, die exklusiv für einen Algorithmus allokiert sind (vgl. z.B. [7]). Allerdings ist SPACE auf Turing-Maschinen definiert und gibt dort nicht die Anzahl der allokierten, sondern der vom Algorithmus besuchten Zellen an. Bei realen Rechnern mit Random Access Memory kann das zu irreführenden Ergebnissen führen. Einige polynomiell begrenzte Probleme können durch riesigen Speicheraufwand in konstanter Zeit erledigt werden. Das Suchen in Mengen kann zum Beispiel in konstanter Zeit ausgeführt werden, wenn eine Hashtabelle mit der Hashfunktion $h(key) = key$ verwendet wird. Diese Tabelle würde jedoch einen unakzeptablen Speicherplatz benötigen, den der Algorithmus zwar allokieren muss, von dem er aber für jedes Problem nur einen winzigen Teil besuchen würde. Für weiterführende Informationen zum Zusammenhang zwischen TIME und SPACE sei an dieser Stelle auf die entsprechend Literatur wie zum Beispiel [5], [22], [7] und [11] verwiesen.
Da fast alle Algorithmen erst einmal das konkrete Problem einlesen müssen, ist der Speicherbedarf für das Problem unabhängig vom Algorithmus. So muss jeder Algorithmus, der zwei Zahlen addieren soll, diese erst einmal einlesen. Daher kann es sinnvoll sein, dass diese Speicherzellen nicht mit gezählt werden (vgl. [22]). Dies entspricht auch der Definition, die hier genutzt werden soll.
Die Laufzeit ist meistens abhängig von einer oder mehreren Problemgrößen. Möchte man zum Beispiel den Mittelwert von einer Menge von Zahlen berechnen, so ist die Anzahl der benötigten Rechenschritte offensichtlich abhängig von der Anzahl der Zahlen, für die der Mittelwert gebildet werden soll. Die Laufzeitkomplexität wird daher immer als Funktion angegeben, die solche abhängige Größen als Parameter hat. Die tatsächliche Laufzeit kann jedoch auch für verschiedene, aber gleich große Probleme variieren. Daher werden bei der Komplexitätsanalyse normalerweise Funktionen angegeben, die eine Abschätzung der Laufzeit darstellen. Dies geschieht hauptsächlich über drei verschiedene Abschätzungen.
Wenn keine formale, sondern eine statistische Untersuchung des Algorithmus vorgenommen wird, wird häufig die durchschnittliche Ausführungszeit angegeben. Eine weitere Möglichkeit ist die bestmögliche Ausführungszeit (Best Case Execution Time - BCET). Sie stellt eine untere Grenze relativ zur Problemgröße dar, die der Algorithmus in keinen Fall unterschreiten kann. Da sie eine Garantie angibt, muss sie formal bestimmt werden. Analog kann auch die schlimmstmögliche Ausführungszeit (Worst Case Execution Time - WCET) angegeben werden. Auch sie stellt eine Garantie dar, und muss daher formal begründet werden. Sie spielt in der formalen Betrachtung von Algorithmen die größte Rolle. Auch bei den Untersuchungen in diesem Kapitel wird das Hauptaugenmerk auf die WCET gelegt.

5.1.2 Asymptotisches Verhalten

Normalerweise ist es nicht so wichtig die exakte WCET eines Algorithmus zu kennen. Die Motivation der Komplexitätstheorie liegt in der Lösung von großen Problemen, da die Laufzeit bei kleinen Problemen üblicherweise akzeptable ist. Bei der Analyse ist daher vor allem das Verhalten bei großen Problemen von Bedeutung. Wenn beispielsweise bekannt ist, dass ein Algorithmus maximal $36n^2 + 3\log_2 n + 47$ Schritte benötigt, so ist der Term n^2 der mit Abstand interessanteste, da er den Term mit der größten Wachstumsrate darstellt. Für große n (also für große Probleme) wird er ausschlaggebend für den Gesamtwert des Terms sein.
Um diese Eigenschaft zu beschreiben wird die O-Notation verwendet, bei der die Komplexität nur über den für große Werte am stärksten wachsenden Term angegeben wird. In [5] wird O folgendermaßen definiert:

Sei $f : \mathbb{N} \to \mathbb{R}^+$ eine Funktion, dann gilt:
$O(f)$ = Menge aller Funktionen $g : \mathbb{N} \to \mathbb{R}^+$, für die es Konstanten $C > 0$ und $n_0 \in \mathbb{N}$ gibt, so dass gilt:

$$g(n) \leq C \cdot f(n) \quad \forall n \geq n_0.$$

Mithilfe dieser Definition lassen sich die Laufzeitfunktionen nun in Klassen von Funktionen einteilen. $O(f)$ sind dann alle Funktionen, die bis auf einen konstanten Faktor C für Eingaben größer als n_0 nicht schneller wachsen als f. Im Zusammenhang mit der Komplexitätsanalyse von Algorithmen spricht man hier von Komplexitätsklassen.
Da es sich bei $O(f)$ um eine Menge von Funktionen handelt, ist die korrekte Schreibweise für das obige Beispiel $36n^2+3\log_2 n+47 \in O(n^2)$, da $C = 100$ und $n_0 = 1$ die nötigen Bedingungen erfüllen. In der Praxis findet sich jedoch auch häufig die Schreibweise $g(n) = O(f(n))$ anstelle von $g(n) \in O(f(n))$. Umgangssprachlich wird meistens auch davon gesprochen, dass ein Algorithmus in $O(f)$ liegt. Gemeint ist dabei, dass die Laufzeitfunktion des Algorithmus ein Element von $O(f)$ ist. [22]
Wichtig ist bei den Komplexitätsklassen, dass sich eine Ordnungsrelation auf ihnen definieren lässt. So gilt zum Beispiel $O(n^2) \subseteq O(n^3)$, da für jede quadratische Funktion g Werte für C und n_0 gefunden werden können, die zeigen, dass g in $O(n^3)$ liegt. Für die Ordnung von einigen Komplexitätsklassen, die in den folgenden Abschnitten von Interesse sind, gilt (vgl. [22]):

$$\begin{aligned} O(\log(\log(n))) &\subset O(\log(n)) \\ &\subset O(n) \\ &\subset O(n \cdot \log(n)) \\ &\subset O(n^2) \\ &\subset O(n^3) \\ &\subset O(2^n) \end{aligned} \tag{5.1}$$

Des Weiteren wird in den nachfolgenden Abschnitten Gebrauch von einigen Rechenregeln gemacht (vgl. [22]):

1. $c \cdot f \in O(f)$ für $c \geq 0$
2. $O(f_1) + O(g) = O(f_1 + g) = O(\max f, g)$

3. $O(f) \cdot O(g) = O(f \cdot g)$

Die ersten beiden Regeln vereinfachen die Handhabung der Komplexitätsfunktionen. Die erste besagt, dass konstante Faktoren bei den Komplexitätsfunktionen nicht berücksichtigt werden müssen. Durch Anwendung der zweiten muss dagegen bei einer Komplexitätsfunktion immer nur der Summand betrachtet werden, der das stärkste Wachstum hat. Damit lässt sich nun formal für das obige Beispiel begründen, dass $36n^2 + 3\log_2 n + 47 \in O(n^2)$.
Die zweite Regel findet zudem häufig Anwendung bei der Komplexitätsanalyse von sequentiellen Programmabschnitten. Hat man zum Beispiel folgenden Programmausschnitt,

```
   T
1 n²  anweisung1();
2  n  anweisung2();
```

bei dem die Komplexität der Anweisung 1 in $O(n^2)$ und die von Anweisung 2 in $O(n)$ liegt, so ergibt sich aufgrund von Regel 2 die Gesamtkomplexitätsfunktion durch $O(n^2) + O(n) = O(n^2 + n) = O(n^2)$.
Wie auch bei diesem Programmausschnitt sollen in den späteren Abschnitten bei der Analyse der Algorithmen die Algorithmen selber abgebildet werden. Vor den Anweisungen wird es dabei immer eine mit T beschriftete Spalte geben, in der die Komplexitätsfunktion der jeweiligen Anweisungen aufgelistet sind. Um die Gesamtkomplexität eines Algorithmus zu bestimmen, müssen dann nur diese Funktionen durch Addition und mithilfe von Regel 2 zusammengefasst werden. T steht dabei für Time, da die Komplexitätsfunktionen das asymptotische Verhalten der Metrik Time beschreiben. Bei dieser Notation werden Bereiche mit verwandter Laufzeit mithilfe von Regel 1 zusammengefasst. Dies lässt sich am folgenden Beispiel illustrieren, bei dem die Komplexitätsfunktionen aller Anweisungen in $O(\mathrm{n})$ liegen.

```
   T
1  │  anweisung1();
2  n  anweisung2();
3  │  anweisung3();
```

Als Gesamtkomplexitätsfunktion ergibt sich also $3n \in O(n)$. Daher werden solchen Bereiche durch einen vertikalen Balken zusammengefasst.
Die letzte Regel findet Anwendung in Schleifen und verschachtelten Funktionen. Im folgenden Programmausschnitt wird die Schleife n-mal durchlaufen. Die Komplexität der Schleife selber liegt daher in $O(n)$. Die Anweisung im Schleifenrumpf hat jedoch eine Komplexität von $O(m)$.

```
     T
1    n  for (i = 1 to n) {
2  n · m    anweisung;
3         }
```

Da die Schleife n-mal ausgeführt wird, liegt die Gesamtlaufzeit aller Aufrufe der Anweisung im Rumpf in $O(n) \cdot O(m) = O(n \cdot m)$. Daher werden die Komplexitätsfunktionen von Anweisungen, die sich innerhalb von Schleifen befinden, immer mit der Komplexitätsfunktion der Schleife multipliziert, bevor sie in die T-Spalte eingetragen werden.

Für die hier betrachteten Algorithmen, hängt die Laufzeit meistens von der Länge der beteiligten Testliste ab. Daher muss auch untersucht werden, wie sich die Längen der Testlisten entwickeln. Daher ist links neben der T-Spalte oftmals eine weiter Spalte S eingetragen. Dort werden die Stellen der Algorithmen markiert, die Auswirkungen auf die Längen der Testlisten haben. Die Werte, die in dieser Spalte eingetragen werden, geben an, im welchen Rahmen sich die Testlisten an dieser Stelle verändern können. Da die Testlisten in den hier betrachteten Algorithmen die einzigen dynamisch wachsenden Strukturen sind, ist diese Spalte mit S wie SPACE beschriftet. Allerdings soll die Notation in dieser Spalte hier nicht so streng definiert werden, da zum Teil mehrere Testlisten an einem Algorithmus beteiligt sind. Die Notizen sollen hier nur aufzeigen, wo und in welcher Art sich die Testlisten innerhalb einer Funktion ändern. Die genauere Beschreibung dieser Stellen erfolgt dann in den begleitenden Texten.

5.1.3 Problem- und Algorithmenkomplexität

Bei den bisherigen Betrachtungen ist immer nach der Laufzeitkomplexität eines gegebenen Algorithmus gefragt. Beim Komplexitätsbegriff ist jedoch darauf zu achten, dass auch häufig von der Komplexität von Problemen gesprochen wird. Ein gängiges Beispiel ist das Sortieren von Arrays. Es gibt Algorithmen, die diese Aufgabe in $O(n^2)$ lösen. Allerdings existieren auch welche, die es in $O(n \log n)$ bewältigen. Man sagt nun, dass ein Problem in $O(f)$ lösbar ist, wenn ein Algorithmus gefunden werden kann, der das Problem eben in $O(f)$ löst.
Auf diese Weise lässt sich nun die Klasse P der polynomiell lösbaren Probleme definieren. Sie enthält alle Probleme, die für ein konstantes k in $O(n^k)$ berechenbar sind. Diese Probleme werden als effizient lösbar betrachtet. Alle anderen Probleme, die nicht in P liegen, werden dagegen als nicht effizient lösbar angesehen. [22]
In dieser Arbeit wird ausschließlich die Algorithmenkomplexität betrachtet.

5.1.4 Stolperfallen

In vielen Fällen lässt sich die Komplexität eines gegebenen Algorithmus einfach durch ein genaues Betrachten der Implementierung ermitteln. Allerdings kann sich die Komplexität selbst in einfachen Algorithmen in scheinbar bedeutungslosen Anweisungen verstecken. Das tritt häufig bei Multiplikationen auf. Ein Prozessor besitzt zwar im Normalfall Multiplikationsbefehle, so dass die meisten Multiplikationen in konstanter Zeit durchgeführt werden können. Allerdings kann das Ergebnis einer Multiplikation doppelt so viel Speicherplatz benötigen, wie die Faktoren.

```
1  int X2Y(int x, int y) {
2    for (i = 1 to y) {
3      x = x * x;
4    }
5    return x;
6  }
```

Die hier abgebildete Funktion berechnet zum Beispiel x^{2^y}. Der Aufbau erweckt den Eindruck, dass die Laufzeit des Algorithmus durch $O(y)$ begrenzt ist. Jedoch wird der Speicherplatz für `x` bereits für relativ kleine `y` schnell verbraucht sein. Da sich die Komplexitätsanalyse für das asymptotische Verhalten bei großen

Eingaben interessiert, muss für eine universelle Implementierung `x` hier durch eine dynamisch wachsende Datenstruktur abgebildet werden. Der daraus resultierende Mehraufwand erfordert eine weitaus höhere Laufzeitabschätzung. [7]
In einigen Fällen ist allerdings nicht der Speicherplatz das Problem, sondern das exponentielle Wachstum des Produkts. Wird es beispielsweise für Schleifenbedingungen verwendet, so treten schnell Fälle auf, in denen auch die Anzahl an Schleifeniterationen exponentiell wächst. Diese Problematik findet sich auch in der Echtzeitanalyse von periodischen Tasksets. Für eine exakte Analyse muss die Hyperperiode des Tasksets untersucht werden, die aus dem kleinsten gemeinsamen Vielfachen aller Taskperioden gebildet wird. Das ist im schlimmsten Fall eben das Produkt aller Perioden.
Die Multiplikations-Problematik zeigt sich auch in den Implementierungen dieser Arbeit. Für die Approximation von Ereignisströmen ist es notwendig, dass Testlisten-Elemente einen Gradientenwert besitzen können, der die Steigung der Ereignisschranken-Funktion angibt. Die Steigung ergibt sich dabei aus dem Quotienten der Periode und der Ausführungszeit des betreffenden Tasks und kann daher exakt durch einen Bruch dargestellt werden. In vielen Algorithmen des Realtime Calculus werden diese Brüche für Multiplikationen und Additionen verwendet. Dazu müssen meist die Nenner beider Brüche multipliziert werden, die dadurch ein exponentielles Wachstum aufweisen können.
Das Problem lässt sich mithilfe von Fließkommazahlen umgehen. Dabei muss jedoch in Kauf genommen werden, dass diese Werte einen kleinen Fehler aufweisen können. Zudem muss mit einer gewissen Vorsicht an die Implementierung gegangen werden. Einige Gesetzmäßigkeiten wie Assoziativ- und Distributivgesetze, die man normalerweise beim Arbeiten mit Zahlen voraussetzt, gelten bei Fließkommazahlen nicht. Daher können Implementierungen, die semantisch scheinbar identisch sind, unterschiedliche Ergebnisse liefern. In einigen Fällen kann der Fehler jedoch durch geschickte Vorgehensweisen minimiert werden.[16]
Um den oben beschriebenen Problemen mit Brüchen auszuweichen, wurden in dieser Arbeit hauptsächlich Gleitkommazahlen verwendet. Der Optimierung der Rechenwege wurde jedoch kein besonderes Augenmerk geschenkt, da es hierzu sehr viele unterschiedliche und zum Teil trickreiche Verfahren gibt, deren Anwendung den Rahmen dieser Arbeit sprengt. Dennoch sollten die Auswirkungen der Gleitkommaanomalien nicht unterschätzt werden, da sie nicht nur zu Rundungsfehlern, sondern auch zu gravierenden semantischen Fehlern führen können, insbesondere wenn es um den Vergleich zweier Zahlen geht. Um die Zuverlässlichkeit der Algorithmen zu garantieren, steht es daher noch aus zu prüfen, ob und in welchen Fällen diese Anomalien Auswirkungen auf das Ergebnis haben.

5.2 Datenstrukturen

Die Testlisten wurden in Kapitel 4 nur als abstrakte Objekte vorgestellt, die eine geordnete Menge von Testlisten-Elementen enthalten. Mathematisch betrachtet besteht eine Menge aus einer Reihe von Objekten, wobei die Zusammensetzung der Menge unveränderbar ist. In der Informatik werden Mengen jedoch meistens durch dynamische Datenstrukturen abgebildet, auf denen verschiedene Operationen definiert sind. Bei den Testlisten ist es zum Beispiel nötig Elemente hinzufügen und löschen zu können oder auch zwei vollständige Testlisten zu addieren.

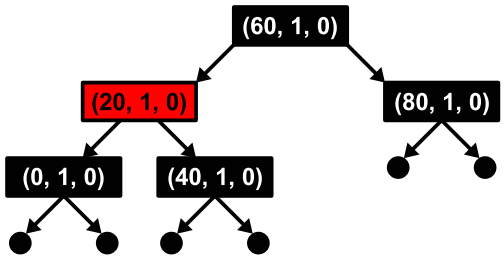


Abbildung 5.1: Testliste als Red-Black-Tree

Für die Komplexitätsbestimmung dieser Operationen ist dabei die Datenstruktur, die der Testliste zugrunde liegt, von entscheidender Bedeutung. [10]
Bei der hier vorgenommenen Implementierung in Java wurde die Testlisten-Klasse von der Klasse `TreeSet` abgeleitet. Die zugrunde liegende Struktur ist ein Red-Black-Tree, der beispielhaft in Abbildung 5.1 dargestellt ist. Ein Red-Black-Tree ist ein binärer Suchbaum, dessen Knoten eine Farbe als zusätzliches Attribut besitzen. In einem solchen Baum werden die Elemente nur in den internen Knoten, nicht aber in den Blättern gespeichert. Er muss zudem fünf weitere Eigenschaften erfüllen:

1. Jeder Knoten ist entweder rot oder schwarz.
2. Die Wurzel ist schwarz.
3. Jedes Blatt ist schwarz.
4. Die Kinder eines roten Knotens sind immer schwarz.
5. Für jeden Knoten enthalten alle Pfade von diesem Knoten zu einem Blatt die gleiche Anzahl von schwarzen Knoten.

Weitere Informationen zu Red-Black-Trees finden sich z.B. in [10]. Entscheidend an dieser Stelle ist, dass das Suchen, Hinzufügen und Entfernen von Elementen in einer solchen Struktur in $O(\log n)$ realisiert werden kann. Das Durchlaufen aller Elemente (Traversieren) in der Reihenfolge, die durch die Ordnung der Elemente gegeben ist, kann auf jedem Binären Suchbaum in $O(\mathrm{n})$ durchgeführt werden.
Anstelle eines Red-Black-Trees kann auch eine einfach verkettete Liste verwendet werden. Das erscheint allerdings auf den ersten Blick nicht sehr effizient, da die Operationen *Suchen*, *Hinzufügen* und *Entfernen* im allgemeinen Fall jeweils in $O(\mathrm{n})$ liegen. Wenn jedoch wie bei einer FIFO-Queue immer nur das erste Element gelöscht und neue Elemente immer nur am Ende eingefügt werden sollen, so lassen sich diese Algorithmen in $O(1)$ implementieren, wenn eine zusätzliche

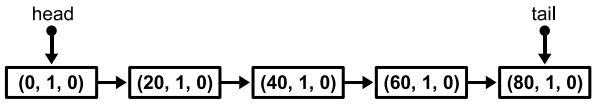


Abbildung 5.2: Testliste als einfach verkettete Liste

Referenz auf das Ende (tail) der Liste mitgeführt wird. Diese Referenz wird für das effiziente Einfügen am Ende der Liste benötigt. Wenn im Folgenden von verketteten Listen die Rede ist, wird daher immer angenommen, dass diese Listen eine solche Referenz besitzen.
Im Vergleich zu den Listen ist eine solche Optimierung beim Red-Black-Tree nicht möglich, da eine Veränderung des Baums immer zur Folge haben kann, dass eine der fünf Bedingungen verletzt wird und der Baum daher umstrukturiert werden muss. Der Aufwand für eine Umstrukturierung liegt jedoch in $O(\log n)$.
Welche Struktur sich nun besser zur Darstellung der Testlisten eignet, hängt also von der Art der Nutzung ab. Daher sind sowohl Red-Black-Trees als auch verkettete Listen die Strukturen, die für die Komplexitätsanalyse hier berücksichtigt werden. Im Folgenden werden deshalb einige Basisoperationen näher betrachtet, die in der Implementierung genutzt werden.

Suchen Bei einem Binärbaum hängt die Komplexität der Suchen-Operation von der Höhe des Baums ab. Die Höhe eines Red-Black-Trees ist immer kleiner oder gleich $2 \cdot \log(n)$, wobei n die Anzahl der Datenelemente ist. Somit kann *Suchen* hier in $O(\log n)$ realisiert werden.
Dagegen müssen bei einer verketteten Liste im schlimmsten Fall alle Elemente überprüft werden. Daher liegt die Komplexität hier in $O(\mathrm{n})$. [10]

Hinzufügen Beim Hinzufügen muss zuerst die richtige Position für das neue Element innerhalb der Testliste gesucht werden. Danach muss das Element dort korrekt eingefügt werden. Da die Testlisten nach den Offsets der Elemente geordnet sind, kann es vorkommen, dass beim Einfügen bereits ein Element mit dem gleichen Offset vorhanden ist. In diesem Fall werden beide Elemente durch Addition vereint:

$$(a, c_1, g_1) + (a, c_2, g_2) = (a, c_1 + c_2, g_1 + g_2)$$

Das geschieht offensichtlich in $O(1)$.
Beim Hinzufügen müssen zwei Fälle unterschieden werden.

Allgemeiner Fall Im allgemeinen Fall, bei dem ein Element an eine beliebige Stelle im Baum eingefügt werden soll, benötigt man für den Red-Black-Tree maximal $2 \cdot \log(n)$ Schritte, um die richtige Stelle zum Einfügen zu finden, und zusätzlich maximal $2 \cdot \log(n)$ Schritte, um den Baum so umzustrukturieren, dass er wieder die Eigenschaften eines Red-Black-Trees erfüllt. Daher liegt die Komplexität hier in $O(\log n)$.[10]
Bei der verketteten Liste benötigt das Finden der richtigen Position maximal n Schritte und das Einfügen selbst kann in $O(1)$ erledigt werden. Somit liegt das *Hinzufügen* hier in $O(\mathrm{n})$. Die allgemeine Form des Hinzufügens wird in den folgenden Programmbeispielen mit `add` angegeben.

Erstes oder letztes Element hinzufügen Da die verkettete Liste Referenzen auf das erste und letzte Element besitzt, kann das Suchen der richtigen Position nun in $O(1)$ erledigt werden. Dadurch liegt die gesamte Operation in $O(1)$. Im Weiteren werden diese Fälle durch `addFirst` bzw. `addLast` gekennzeichnet.

Eine ähnliche Optimierung ist beim Red-Black-Tree nicht möglich, selbst wenn Referenzen auf das erste und letzte Element mitgeführt werden, da das Restrukturieren des Baums immer noch in O(log n) liegt. [10]

Entfernen Das Entfernen erfolgt analog zum Hinzufügen von Elementen. Zuerst muss die Position des betreffenden Elements gefunden werden und danach kann es aus der Struktur entfernt werden. Die Überlegungen zum allgemeinen Fall und zu den beiden Spezialfällen treffen auch hier zu. Eine Ausnahme ist jedoch das Entfernen des letzten Elements aus der verketteten Liste. Diese Operation benötigt immer n Schritte, da die Liste einmal vollständig iteriert werden muss, um die Referenz auf das neue letzte Element zu setzen. Das kann vermieden werden, indem man eine doppelt verkettete Liste verwendet. Da diese Operation jedoch in den hier betrachteten Algorithmen nicht verwendet wird, reicht hier eine einfach verkettete Liste.

Die hier beschriebenen Operationen beschäftigen sich jeweils mit nur einem Element der Testliste. Es lassen sich jedoch auch Operationen auf der gesamten Testliste definieren, wie zum Beispiel die Vereinigung zweier Testlisten. Für solche Operationen ist nun interessant zu sehen, dass eine Testliste, die durch eine einfach verkettete Liste dargestellt wird, in O(n) in einen Red-Black-Tree umgewandelt werden kann und umgekehrt.
In [17] wurde ein Algorithmus vorgestellt, der beliebige Binärbäume in O(n) in ausbalancierte Binärbäume umwandelt und dabei nur eine konstante Menge an zusätzlichem Speicher benötigt. Die Vorgehensweise ist dabei in zwei Schritte unterteilt, die in Abbildung 5.3 dargestellt sind. Beide Schritte können jeweils in $O(n)$ durchgeführt werden. Im ersten Schritt wird aus einem beliebigen Binärbaum eine Vine-Struktur erzeugt. Das ist ein Binärbaum, dessen Knoten immer nur einen Kindknoten haben. Die Vine-Struktur entspricht im Prinzip einer einfach verketteten Liste. Da ein Red-Black-Tree ein Binärbaum ist, kann auf diese Weise mit kleinen Modifikationen eine Testliste in O(n) von einem Red-Black-Tree in eine einfach verkettete Liste transformiert werden.
Mit dem zweiten Schritt kann eine einfach verkettete Liste in einen ausbalancier-

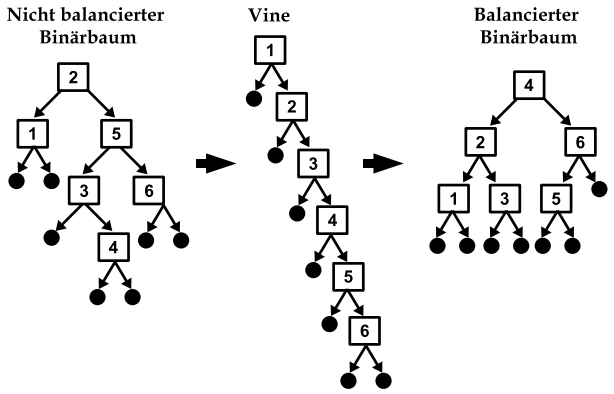


Abbildung 5.3: Ausbalancieren eines Binärbaumes in $O(n)$

ten Binärbaum umgewandelt werden. Färbt man nun alle Knoten der tiefsten Ebene rot und alle anderen schwarz, so erfüllt der Baum zudem die Eigenschaften eines Red-Black-Trees. Es ist leicht zu sehen, dass auch das Einfärben in O(n) bewerkstelligt werden kann. Daher kann eine einfach verkettete List in $O(n)$ in einen Red-Black-Tree transformiert werden.
Aufgrund dieser Beobachtung kann für die Komplexitätsanalyse von Operationen, deren Eingabe aus Testlisten besteht, immer die geeignetste Art der Struktur gewählt werden, solange der betrachtete Algorithmus in O(n) liegt.

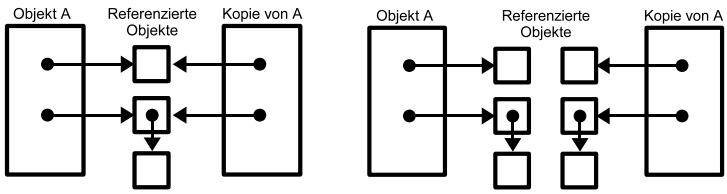
5.3 Operationen

5.3.1 Traversieren

Traversieren bezeichnet das Durchlaufen aller Knoten eines Graphen. Bei Graphen, auf deren Knoten eine Ordnung definiert ist, macht das Traversieren entlang dieser Ordnung am meisten Sinn. Bei verketteten Listen ist leicht ersichtlich, das dies in O(n) zu bewerkstelligen ist, indem einfach allen Referenzen gefolgt wird. Dabei ist n die Länge der Liste.
Auch Binärbäume können in O(n) traversiert werden. Dazu gibt es verschiedene Möglichkeiten. Die einfachste ist, dies mit einer rekursiven Funktion zu machen. Allerdings wird das Traversieren der Testlisten meistens im Zusammenhang mit dem Durchlaufen einer Schleife verwendet, bei der in jeder Iteration ein Element der Liste von Interesse ist. Eine rekursive Funktion ist hier umständlich, da die rekursive Funktion bei jedem Aufruf angehalten werden muss, um sich mit der Schleife zu synchronisieren.
Eine nicht rekursive Lösung besteht zum Beispiel darin, in jedem Knoten eine zusätzliche Referenz auf den Vaterknoten einzufügen. Dann kann die Iteration einfach durch ein geschicktes Verfolgen der Referenzen bewerkstelligt werden. Dafür werden jedoch n zusätzliche Referenzen benötigt.
Eine weitere Möglichkeit ist, ein Stack-basiertes Verfahren zu verwenden, bei dem sich für jede Ebene gemerkt wird, ob man sich gerade im linken oder rechten Teilbaum befindet. Die Größe des Stack entspricht dabei der Höhe des Baums, die bei einem Red-Black-Tree $\log(n)$ beträgt.
Testlisten können also in linearer Zeit sowohl auf verketteten Listen, als auch auf Red-Black-Trees traversiert werden, wobei jedoch der zusätzliche Speicheraufwand bei Red-Black-Trees in $O(\log n)$ liegt. Bei allen folgenden Programmbeispielen werden daher alle `for`-Schleifenköpfe, die in der Java-Syntax `for (type variable: <Iterable>)` vorliegen, mit linearer Laufzeit markiert.

5.3.2 Kopieren

Beim Kopieren (oder auch Klonen) von Objekten unterscheidet man zwischen tiefem und oberflächlichem Kopieren. Beim oberflächlichen Kopieren wird nur das Objekt selbst kopiert, nicht aber die Objekte, die es referenziert. Dabei ist die Kopie durch die referenzierten Objekte immer noch abhängig vom Original (s. Abbildung 5.4(a)). Für die Testlisten wird jedoch ein tiefes Kopieren benötigt, bei dem die Testlisten-Elemente mit kopiert werden. Daher müssen sowohl die Struktur der Testliste als auch alle Testlistenelemente kopiert werden. Beides kann sowohl für verkettet Listen, als auch für Red-Black-Trees während des



(a) Oberflächliches Kopieren (b) Tiefes Kopieren

Abbildung 5.4: Varianten beim Klonen

Traversierens geschehen und liegt daher in O(n).

5.3.3 Vereinen

Das Vereinen zweier Testlisten ist mit einer verketteten Liste in $O(n+m)$ ohne zusätzlichen Speicher möglich, wobei n die Länge der einen Liste und m die der anderen ist. Dazu werden beide Listen gleichzeitig durchlaufen. Währenddessen werden die Referenzen der Elemente so geändert, dass sie der Ordnung beider Listen folgen.
Da Binärbäume mit linearem Zeitaufwand in verkettete Listen transformiert werden können, gilt dies auch für Red-Black-Trees.
Betrachtet man die Funktion, die durch eine Testliste beschrieben wird, so stellt die Vereinigung die Addition der Funktionen dar. In den Programmbeispielen wird die Vereinigung daher mit `addTestList` bezeichnet. Die resultierende Liste hat eine maximale Länge von $n+m$.

5.3.4 Negation

Das Negieren einer Testliste entspricht dem Spiegeln der dargestellten Funktion an der X-Achse. Dazu werden einfach die Gradienten und Kosten aller Testlisten-Element negiert. Anhand der abgebildeten Funktion ist leicht zu erkennen, dass diese Operation in O(n) liegt.

```
  T
1    public void negate() {
2  n   for (TestListElement tle : this) {
3  n     tle.negate();
4      }
5    }
```

Listing 5.1: TestList.negate()

5.3.5 Skalierung

Durch die Skalierung wird jeder Wert der dargestellten Funktion mit einem konstanten Faktor multipliziert. Dies wird benötigt, um aus der Beschreibung der Ereignisse die durch sie angeforderte Prozessorzeit zu ermitteln. Dabei wird der Ereignisstrom bzw. die ihn darstellende Testliste mit der Ausführungszeit des Tasks skaliert, der durch diesen Strom aktiviert wird. Auch hier ist durch das Listing 5.2 leicht ersichtlich, dass die Funktion in O(n) realisierbar ist.

```
   T
1    public void scale(double factor){
2  n   for (TestListElement tle : this) {
3  n     tle.scale(factor);
4      }
5    }
```

Listing 5.2: TestList.scale

5.3.6 Shift

Durch die Shiftoperation wird der Funktionsgraph der Funktion entlang der X-Achse verschoben. Es wird dabei jedes Element der Liste geshiftet, indem der Offset jeweils mit dem Wert addiert wird, um den die Funktion verschoben werden soll. Diese Funktion ist ebenfalls in O(n) realisierbar.

```
   T
1    public void shift() {
2  n   for (TestListElement tle : this) {
3  n     tle.shift();
4      }
5    }
```

Listing 5.3: TestList.shift

5.3.7 Lift

Analog zur `shift`-Funktion kann die Testliste auch entlang der Y-Achse verschoben werden. Dies geschieht dadurch, dass zu Beginn der Testliste ein Element hinzugefügt wird, das die Menge an Kosten erzeugt, um die die Liste verschoben werden soll.
Da das Hinzufügen mittels `addFirst` geschieht, kann `lift` für verkettete Listen in $O(1)$ und für Red-Black-Trees in O(log n) durchgeführt werden. Zudem ist zu beachten, dass die Testliste um eine Element verlängert wird.

```
  S T
1    public void lift(double lift){
2 +1 1   this.addFirst(new TestListElement(0, lift,0));
3    }
```

Listing 5.4: TestList.lift

5.3.8 Subtraktion

Die Subtraktion zweier Funktionen kann durch die Negation des Subtrahenden und anschließender Addition zum Minuenden erreicht werden. So ist auch das Vorgehen bei den Testlisten. Das folgende Programmbeispiel zeigt, dass die Komplexität dabei in O(n+m) liegt, wobei n die Länge des Subtrahenden und m die Länge des Minuenden ist.
Da die Negation die Länge der Liste nicht ändert, entspricht die maximale Länge des Ergebnisses mit $n + m$ Elementen der maximal durch die Addition erreichbaren Länge.

```
    S     T
1           public void subTestList(TestList subtrahend) {
2           n   negate(subtrahend);
3 n+m  n+m     this.addTestList(subtrahend);
4           }
```

Listing 5.5: TestList.subTestList()

5.3.9 Minimum

```
   S  T
1        public static TestList min(TestList a, TestList b) {

3          if (a.isEmpty() || b.isEmpty())  return new TestList();

5          TestList result = new TestList();
6          int act, min = A;
7        1 TestListElement tle;
8          double IOld = 0,  intersect = Double.POSITIVE_INFINITY;
9          double[] c = {0, 0, 0}; //current costs of list A, B and R(resulting list)
10         double[] g = {0, 0, 0}; //current gradient of list A, B and R
11         Iterator<TestListElement>[] iter = {a.iterator(), b.iterator()};
12         TestListElement next[] = {iter[A].next(), iter[B].next()};

14         while ((next[A] != null) || (next[B] != null)) {

16           act = first(next);  // decide which testlist has the next element
17           tle = next[act];    // select next element
18           next[act] = iter[act].hasNext() ? iter[act].next() : null;

20           if (intersect < tle.getOffset()) {  // Input lists intersect
21             if (g[R] != g[A]) min = A;  // Select the new minimum list
22             else min = B;               //
23 n+m       // add interesection point
24  -1 ●       result.addLast(new TestListElement(intersect, 0, g[min] - g[R]));
25             // update cost and gradient values up to the intersection point
26             c[A] += (intersect - IOld) * g[A];
27             c[B] += (intersect - IOld) * g[B];
28             c[R] = c[min];
29             g[R] = g[min];
30             IOld = intersect;
31             intersect = Double.POSITIVE_INFINITY;
32           }
33           // update cost and gradient values
34           c[act] += tle.getCosts();
35           c[A] += (tle.getOffset() - IOld) * g[A];
36           c[B] += (tle.getOffset() - IOld) * g[B];
37     n+m   c[R] += (tle.getOffset() - IOld) * g[R];
38           g[act] += tle.getGradient();

40           if (c[A] > c[B]) min = B;   // select the minimum list
41           else min = A;               //
42           // If the current element is a minimum, add it.
43           if ((c[min] != c[R]) || (g[min] != g[R])) {
44 n+m ●       result.addLast(new TestListElement(tle.getOffset(),
45                                                c[min] - c[R],
46                                                g[min] - g[R]));
47           }
48           // Update cost and gradient of the resulting list
49           c[R] = c[min];
50           g[R] = g[min];

52           // Check if an intersection might occure
53           if (g[not(min)] < g[min]) {
54             intersect = (c[not(min)]-c[min])/(g[min]-g[not(min)]) + tle.getOffset();
55           } else {
56             intersect = Double.POSITIVE_INFINITY;
57           }

59           IOld = tle.getOffset();
60         }
61         // If an intersection occurs after the last elements of both
62         // input list, add the intersection point
63         if (intersect < Double.POSITIVE_INFINITY) {
64   1 ●     result.addLast(new TestListElement(intersect, 0, g[not(min)] - g[min]));
65       1 }
66         return result;
67       }
```

Listing 5.6: TestList.min

Der Algorithmus zur Berechnung des Minimums zweier Testlisten sollte mit Sorgfalt betrachtet werden, da er bei Berechnung der Ankunftskurven sehr häufig aufgerufen wird. Er ist in Listing 5.6 abgebildet.
Als Struktur der Testliste bietet sich hier eine einfach verkettete Liste an, da die einzig hierbei relevante Funktion die `addLast`-Funktion in Zeile 24, 44 und 64 ist. Durch Verwendung einer verketteten Liste kann sie in $O(1)$ durchgeführt werden.
Die Funktion besitzt eine `while`-Schleife. Aus den Zeilen 11-18 ist ersichtlich, dass diese für jedes Element beider Eingabelisten einmal ausgeführt wird. Da die WCET aller Anweisung innerhalb der Schleife konstant ist, kann die Laufzeit der Funktion mit O(n+m) abgeschätzt werden.
Für die Länge der Ergebnisliste sind die `addLast`-Aufrufe verantwortlich. Der Aufruf in Zeile 44 kann prinzipiell in jedem Schleifendurchlauf geschehen, wohingegen der in Zeile 24 aufgrund der initialen Belegung von `intersect` nicht im ersten Durchlauf aufgerufen werden kann (vgl. Z.8 und 20). Dafür ist jedoch ein weiterer Aufruf am Ende der Funktion möglich. Daher kann die Ergebnisliste eine maximale Länge von $2(n + m)$ Elementen haben.
Damit ein solcher Fall auftritt, muss in jedem Schleifendurchlauf ein Schnitt der beiden Eingabe-Testlisten erkannt werden und zugleich die Testliste des aktuellen Elements an dieser Stelle das Minimum sein. Wie in Abbildung 5.5 zu sehen ist, lässt sich ein solches Verhalten mit zwei konvexen Kurven oder unter der Verwendung von Stufenfunktionen erzeugen. Da die dort abgebildeten Kurven gültige, untere Service- oder Ankunftskurven darstellen, muss auch in der Praxis mit der Maximallänge gerechnet werden.

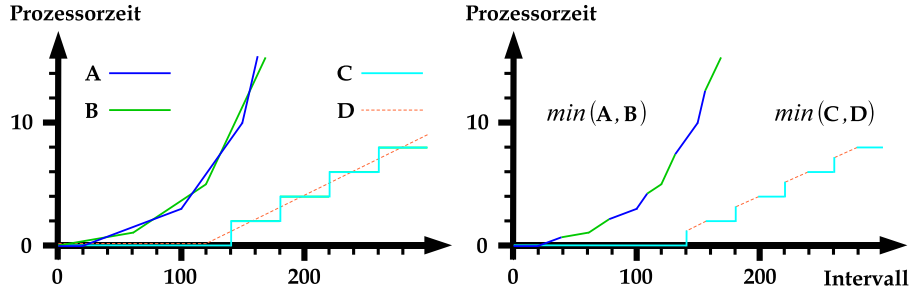


Abbildung 5.5: Erzeugung maximaler Testlistenlängen bei der Minimum-Operation durch konvexe Funktionen (A,B) und Treppenfunktionen (C,D)

5.3.10 Maximum

Die Berechnung des Maximums zweier Kurven ist sehr ähnlich aufgebaut wie die Minimumberechnung (s. Lisiting 5.7). Es gibt zwar eine `addLast`-Operation mehr, als bei der Minimumberechnung, da sich die beiden in Zeile 35 und 38 jedoch gegenseitig ausschließen, kann durch sie in jedem Durchlauf insgesamt nur ein Element der Liste hinzugefügt werden. Daher erzeugen sie zusammen auch nur maximal $n + m$ Elemente.
Es ergibt sich also auch hier eine maximale Länge von $2(n + m)$ für die resultierende Testliste und ein asymptotisches Verhalten von O(n+m) für die WCET.
Es ist leicht ersichtlich, dass sich analog zu den Testlisten in Abbildung 5.5 für

die Maximum-Funktion sehr einfach lange Ergebnislisten durch die Verwendung von konkaven Funktionen oder Stufenfunktionen erzielen lassen.

```
    S  T
1       public static TestList max(TestList a, TestList b) {

3         TestList result = new TestList();
4         int act, max = A;
5         TestListElement tle;
6         double IOld = 0,  intersect = Double.POSITIVE_INFINITY;
7         double[] c = {0, 0};     //current costs of list A and B
8       1 double[] g = {0, 0};     //current gradients of list A and B
9         Iterator<TestListElement>[] iter = {a.iterator(), b.iterator()};
10        TestListElement next[] = {null, null};
11        if (iter[A].hasNext()) next[A] = iter[A].next();
12        if (iter[B].hasNext()) next[B] = iter[B].next();

14        while ((next[A] != null) || (next[B] != null)) {

16          act = first(next);   // select the testlist with next element
17          tle = next[act];     // get the next element
18          next[act] = iter[act].hasNext() ? iter[act].next() : null;

20          // handle intersections
21          if (intersect < tle.getOffset()) {
22  n+m     result.addLast(new TestListElement(intersect, 0, g[not(max)] - g[max]));
23   -1       max = not(max);
24          }
25          intersect = Double.POSITIVE_INFINITY;

27          //update costs and gradient values
28          c[A] += (tle.getOffset() - IOld) * g[A];
29          c[B] += (tle.getOffset() - IOld) * g[B];
30    n+m   c[act] += tle.getCosts();
31          g[act] +=tle.getGradient();

33          // add this element, if it is maximum
34          if (act == max) {
35            result.addLast((TestListElement)tle.clone());
36  n+m     } else {
37            if (c[act] >= c[not(act)]) {
38              result.addLast(new TestListElement(tle.getOffset(),
39                                                 c[act] - c[not(act)],
40                                                 g[act] - g[not(act)]));
41              max = act;
42            }
43          }

45          // calculate expected intersection point
46          if ((g[max] < g[not(max)])) {
47            intersect = tle.getOffset()+(c[max]-c[not(max)])/(g[not(max)]-g[max]);
48          }
49          IOld = tle.getOffset();
50        }

52        // add intersection poiunts, that occure after the last elements
53        if (intersect < Double.POSITIVE_INFINITY) {
54  1       result.addLast(new TestListElement(intersect, 0, g[not(max)]-g[max]));
55        }
56        return result;
57      }
```

Listing 5.7: TestList.max

5.3.11 Minimum Split

Bisher war die Komplexitätsbestimmung noch relativ einfach. Das ändert sich jetzt jedoch bei der `MinimumSplit`-Operation, die in Listing 5.8 dargestellt ist.

```
   S     T
1          public static TestList minimumSplit(TestList a, TestList b) {

3            if ((a.isEmpty() || b.isEmpty()) return new TestList();

5            int act;
6            double shift;
7            double[] costs = {0, 0};
8            double[] gradient = {0, 0};
9         1  double[] IOld = {0, 0};
10           TestListElement tle;
11           TestList result = new TestList();
12           TestList[] tl = {a, b};
13           Iterator<TestListElement>[] iter = {a.iterator(), b.iterator()};
14           TestListElement next[] = {iter[A].next(), iter[B].next()};
15    n+m    TestList[] tlShiftClone = {TestList)a.clone(), (TestList)b.clone()};

17           //initialize result testlist
18 +1●  1    result.add(new TestListElement(0, 0, 1));

20    n+m    while ((next[A] != null) || (next[B] != null)) {

22             act = first(next);   // decide which testlist has the next element
23             tle = next[act];     // get next element
24    n+m      next[act] = iter[act].hasNext() ? iter[act].next() : null;

26             shift = tle.getOffset() - IOld[act];
27             costs[act] += shift * gradient[act];

29             // shift the other testlist to the current x-position
30    2nm      tlShiftClone[not(act)].shift(shift);
31             // lift the other testlist to the current y-value
32 +1●n+m     tlShiftClone[not(act)].lift(costs[act]);
33             // calculate minimum
34 *1●  *2     result = TestList.min(result, tlShiftClone[not(act)]);
35             // unlift the other testlist
36 -1●         tlShiftClone[not(act)].lift(-costs[act]);

38    n+m      costs[act] += tle.getCosts();
39             gradient[act] += tle.getGradient();
40             IOld[act] = tle.getOffset();
41           }
42           return result;
43         }
```

$*^1 = (\log_2(mn) + 3) \cdot mn$

$*^2 = ((\log_2 m)^2 + (\log_2 n)^2) \cdot mn$

Listing 5.8: TestList.minimumSplit

Die Testlisten werden hier durch verkettete Listen dargestellt, da dies für die `lift`-Funktionen in Zeile 32 und 36 und für die Minimumfunktion in Zeile 34 günstiger ist (s. Abschnitt 5.3.7 und 5.3.9).
Auch diese Funktion enthält lediglich eine Schleife, die über die Elemente beider Listen iteriert und daher $n+m$ mal ausgeführt wird, wobei n und m die Längen der beiden Eingabelisten sind. Bis auf die `shift`- und die `min`-Anweisungen in Zeile 30 und 34 haben alle Anweisungen des Schleifenrumpfes eine konstante WCET.
Bei der `shift`-Anweisung wird immer eine der Eingabelisten geshiftet und zwar immer diejenige, zu der das aktuelle Iterations-Element nicht gehört. Daher wird

die Liste mit den m Elementen n mal geshiftet und die mit den n Elementen m mal. Insgesamt ergibt sich daher für diese Funktion eine Laufzeitabschätzung von $O(2\mathrm{nm})$.
Schwieriger wird die Abschätzung für die Minimum-Funktion. Relevant hierfür sind die Längen der Eingabelisten für diese Funktion. Eine davon ist die geshiftete Liste (`tlShiftClone`), die entweder eine Länge von m oder n hat. Die andere ist jeweils das Resultat des vorherigen Schleifendurchlaufs (`result`). An dieser Stelle wird also eine obere Grenze für die Länge von `result` in jedem Schleifendurchlauf benötigt.
Bevor diese ermittelt wird, soll hier kurz etwas zu den Zeilen 18, 32 und 36 gesagt werden. Sie sind in der S-Spalte markiert, da auch sie Einfluss auf die Längen der beteiligten Testlisten haben. In Zeile 18 wird die `result`-Liste mit einem einzelnen Element initialisiert. Da dies vor dem Schleifenrumpf geschieht, hat diese Anweisung kaum Einfluss auf die weitere Länge von `result`. In den Zeilen 32 und 36 wird durch die `lift`-Anweisung jeweils ein Element am Anfang der Liste eingefügt. Da beide Elemente jedoch den gleichen Offset haben, werden sie bei der zweiten `lift`-Anweisung durch:

$$(0, 0, \texttt{costs[act]}) \; + \; (0, 0, -\texttt{costs[act]}) \; = \; (0, 0, 0)$$

zu einem Element kombiniert, das keinen Effekt hat und daher wieder aus der Liste gelöscht wird. Im Endeffekt verändern sich daher die Längen der `tlShiftClone`-Listen durch die Schleifeniterationen nicht. Daher hat hier hauptsächlich die Minimumfunktion Auswirkungen auf die Länge von `result`.
In Abschnitt 5.3.9 wurde gezeigt, dass bei einer Minimum-Operation von zwei Testlisten der Länge s und t die Länge der Ergebnisliste $2(s+t)$ nicht überschreitet. Diese Rechnung kann man einfach für alle Iterationen fortführen. Jedoch ist nicht bekannt, in welcher Reihenfolge die Elemente der beiden Eingabelisten von `MinimumSplit` abgearbeitet werden. Daher wird die Länge des zweiten Operanden der Minimum-Funktion in der i-ten Iteration der `while`-Schleife hier mit l_i bezeichnet, wobei l_i entweder n oder m ist. Die maximale Länge von `result` sieht daher für die einzelnen Iterationen wie folgt aus:

$$\begin{array}{rrcl}
1. & l_1 & = & 2^0 \cdot l_1 \\
2. & 2(l_1 + l_2) & = & 2^1 \cdot l_1 + 2^1 \cdot l_2 \\
3. & 2(2(l_1 + l_2) + l_3) & = & 2^2 \cdot l_1 + 2^2 \cdot l_2 + 2^1 \cdot l_3 \\
4. & 2(2(2(l_1 + l_2) + l_3) + l_4) & = & 2^3 \cdot l_1 + 2^3 \cdot l_2 + 2^2 \cdot l_3 + 2^1 \cdot l_4 \\
\vdots & \vdots & = & \vdots \\
n+m. & \cdots & = & 2^{n+m-1} \cdot l_1 + 2^{n+m-1} \cdot l_2 + \cdots + 2^1 \cdot l_{n+m}
\end{array}$$

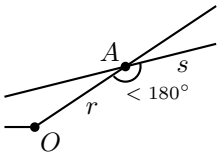
Am Term der letzten Iteration sieht man, dass dieser am kleinsten ist, wenn für die l_i's mit kleinem Index der kleinere Wert von n und m gewählt wird und für die höheren Indizes genau umgekehrt. Da die Reihenfolge, in der die Testlisten für die Minimum-Berechnung verwendet werden, für das Ergebnis irrelevant ist, kann der Algorithmus im Prinzip die optimale Reihenfolge selber wählen. Insbesondere kann er zuerst die kleineren Listen für die Minimum-Berechnung verwenden. Eine solche Implementierung verändert die Komplexitätsklasse des Algorithmus nicht und soll daher für diese Berechnung angenommen werden. Daher ergibt sich nach $n+m$ Iterationen folgende Maximallänge ($r = \min(n, m)$

und $s = \max(m, n)$):

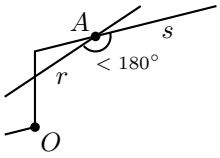
$$\begin{aligned} & 2^{r+s-1} \cdot r + 2^{r+s-1} \cdot r + 2^{r+s-2} \cdot r + \cdots + 2^{r+1} \cdot r + 2^r \cdot s + \cdots + 2^1 \cdot s \\ = \; & (2^{r+s-1} + (2^{r+s} - 2) - (2^{r+1} - 2)) \cdot r + (2^{r+1} - 2) \cdot s \\ = \; & (2^{s-1} + 2^s - 2) \cdot 2^r \cdot r + (2^{r+1} - 2) \cdot s \end{aligned}$$

Hier lohnt sich jedoch ein etwas genauerer Blick, da man ein exponentielles Wachstum selbst im schlimmsten Fall intuitiv nicht erwarten würde. Dazu muss noch einmal die Minimum-Funktion betrachtet werden.
Dort wurde gezeigt, dass die Ergebnisliste nur dann doppelt so lang sein kann wie beide Eingabelisten, wenn jedes Element im Ergebnis vorhanden ist und zusätzlich jedes Element für einen Schnittpunkt der beiden Listen „verantwortlich“ ist. Das meint, dass während jedes Iterationsschrittes der Minimum-Funktion `intersect` auf einen Wert kleiner als der Offset des nächsten Elements gesetzt wird. Es gehören also bei der Minimumfunktion im Graphen der Ergebnisliste zu jedem Element der Eingangslisten maximal zwei Punkte. Das ist zum Einen der Originalpunkt O und der zusätzliche Punkt A, der durch den Schnitt entsteht (s. untere Skizzen). Da `intersect` einen kleineren Wert als der nächste Offset haben muss, damit es zu einem Schnitt kommt, folgt das Testlisten-Element für A in der Ergebnisliste immer dem für O. Bei Elementen, deren Kostenwert größer als Null ist, können eigentlich drei Punkte auftreten, was im Graphen jeweils durch ein Segment mit unendlicher Steigung zu sehen ist (Skizze b). Da für die Minimum-Überlegungen nur die kleinsten Werte von Bedeutung sind, ist hier mit dem Originalpunkt immer der kleinste Wert an einer solchen Steigung gemeint.

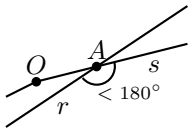
a)



b)



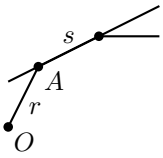
c)



Von den beiden Punkten ist an dieser Stelle vor allem der Schnittpunkt A interessant. Dieser ist die Ursache des exponentiellen Wachstums in der obigen Gleichung. Denn jeder Schnittpunkt kann in jeder weiteren Schleifeniteration neue Schnittpunkte erzeugen, die wiederum in jeder weiteren Iteration Schnittpunkte erzeugen können, die wiederum .... Dass das schnell zu einer Plage wird, kennt man ja von Mäusepopulationen.
Im Folgenden wird nun gezeigt, dass ein Schnittpunkt entweder keinen weiteren Schnittpunkt erzeugen kann, oder jedesmal wenn er einen erzeugt, wird an einer anderen Stelle ein Punkt zwangsweise aus dieser Rechnung fallen.

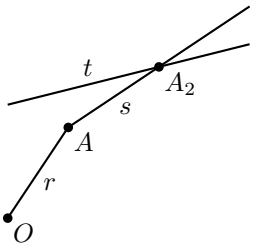
Bei jedem Schnittpunkt A hat das Segment r vor diesem Punkt in der Ergebnisfunktion eine größere Steigung, als das nachfolgende Segment s. Ein Schnittpunkt erzeugt daher eine konkave Ecke. Zudem ist der Kostenwert eines Testlisten-Elements für einen Schnittpunkt immer Null.
Im nächsten Iterationsschritt sind nun drei Fälle zu unterscheiden. Entweder erzeugt der Schnittpunkt A einen weiteren Schnittpunkt A_2 im Segment s, oder er erzeugt keinen, oder der andere Operand der Minimum-Operation hat ein Segment, das auf einem Intervall deckungsgleich mit s ist.
Wird kein weiterer Schnittpunkt erzeugt, so fällt dieser auch aus der Rechnung raus.

In dem Fall, in dem s deckungsgleich mit einem anderen Segment ist, ist leicht ersichtlich, dass hier nur die Anfangs- und Endpunkte der beteiligten Segmente relevant sind und daher auch hier der nicht existente Schnittpunkt aus der Rechnung rausfällt. Ein Beispiel hierzu ist in der Skizze rechts zu sehen.

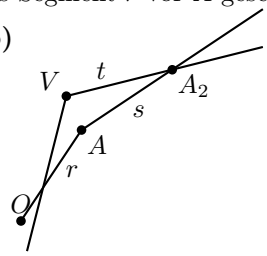


Im verbleibenden Fall wird A_2 durch A erzeugt. Das bedeutet, dass zwischen A und A_2 kein weiterer Punkt liegt und das schneidende Segment t eine kleinere Steigung als das Segment s hat (s. untere Skizzen). Insgesamt gilt für die Steigungen g der Segmente: $g_r > g_s > g_t$. Nun sind wieder zwei Fälle zu unterscheiden. Entweder wird das Segment r vor A geschnitten oder nicht.

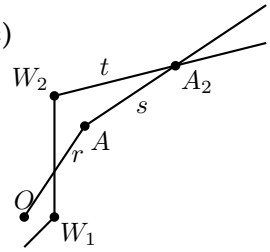
a)



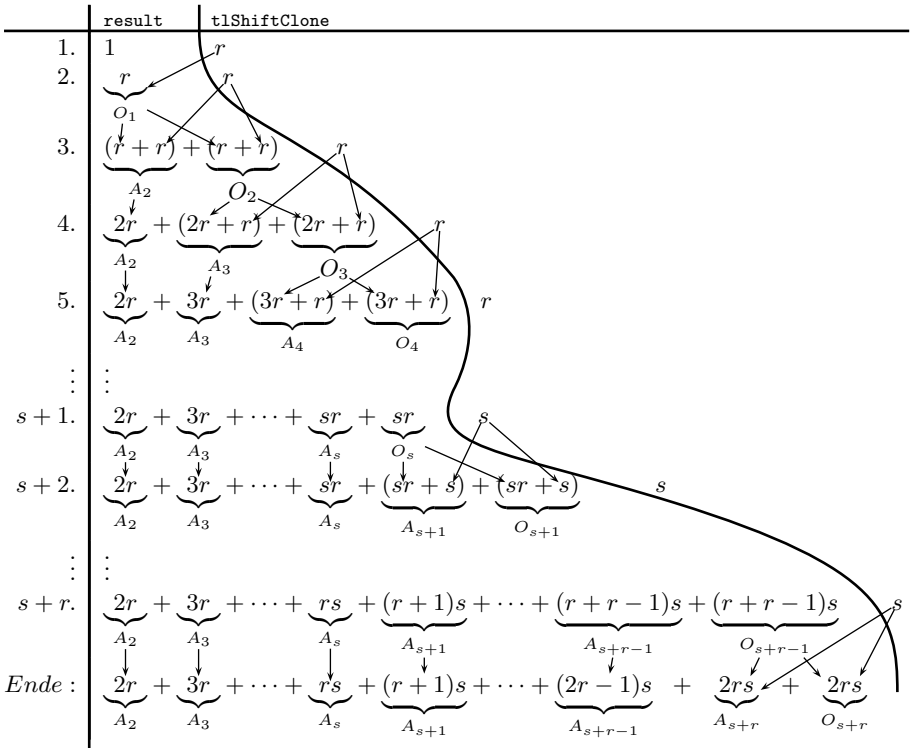
b)



c)



Wird es nicht geschnitten, so erzeugt der Anfangspunkt dieses Segmentes keinen Schnittpunkt, der somit aus der Rechnung fällt (s. Skizze a).
Wird es jedoch geschnitten, so kann dies durch ein Segment mit endlicher (s. Skizze b) oder unendlicher (s. Skizze c) Steigung verursacht werden. Geschieht dies durch ein endliches Segment, so kann es wegen $g_r > g_t$ nicht t sein. Daher muss es einen weiteren Punkt V geben, der im Maximum liegt und somit aus der Rechnung fällt. Da A und A_2 direkt aufeinander folgen, liegt V vor A (s. Skizze b).
Wird der Schnitt jedoch durch ein unendliches Segment erzeugt, so werden die beiden Punkte W_1 und W_2 aufgrund der unendlichen Steigung von w in der Ergebnisliste durch ein Testlistenelement W dargestellt. Daher fehlt zwischen W und A der zusätzliche Schnittpunkt, der einen Offset größer als den von W haben müsste. In diesem Fall fällt der zusätzliche Schnittpunkt nach W somit aus der Rechnung.
Damit ist gezeigt, dass ein Schnittpunkt im nächsten Iterationsschritt in der Summe kein weiteres Testlistenelement erzeugen kann. Da das allein auf der Tatsache beruht, dass ein Schnitt immer eine konkave Ecke erzeugt und das natürlich in allen weiteren Iterationen so bleibt, kann ein Schnittpunkt in allen weiteren Iterationen nicht dazu führen, dass sich die Ergebnisliste verlängert.
Betrachtet man vor diesem Hintergrund noch einmal die Berechnung zur Maximallänge von `result`, so entfällt der exponentielle Charakter. Für die Berechnung ist im Folgenden für jeden Iterationsschritt die maximale Länge der `result`- und der aktuellen `tlShiftClone`-Testliste angegeben. Auch hier sollen die Iterationen mit der kürzeren Liste mit r Elementen beginnen. Es gilt dabei wieder $r = \min(n, m)$ und $s = \max(n, m)$.



$$
\begin{aligned}
&= \left(\sum_{i=2}^{s} i\right) r + r^2 s + \left(\sum_{i=1}^{r} i\right) s + 2rs \\
&= \left(\frac{s^2 r}{2} + \frac{sr}{2} - r\right) + r^2 s + \left(\frac{r^2 s}{2} + \frac{rs}{2}\right) + 2rs \\
&= \frac{1}{2} s^2 r + \frac{3}{2} r^2 s + 3rs - r \qquad (5.2)
\end{aligned}
$$

In jedem Schritt ist jeweils eingetragen, wie viele Originalpunkte aus den Eingabelisten aktuell maximal in `result` enthalten sein können. Diese Anzahl ist für den i-ten Schritt durch O_i markiert. Zudem ist jeweils die maximal mögliche Anzahl an erzeugten Schnittpunkten eingetragen. Dabei bezeichnet A_i die Anzahl an Schnittpunkten, die im i-ten Schritt erzeugt werden können. Schnittpunkte beschreiben immer eine konkave Ecke. Bei dieser etwas genaueren Betrachtung zeigt sich nun, dass das Wachstum der Liste polynomiell begrenzt ist.
Es kann jedoch eine noch kleinere Grenze gefunden werden. Die Funktionsweise der `MinimumSplit`-Funktion beruht darauf, dass die Funktion einer Testliste an ihrem Ursprung entlang der Funktion der anderen Testliste verschoben wird. In Abbildung 5.6 sind auf der linken Seite zwei Testlisten dargestellt, für die die `MinimumSplit`-Funktion ausgeführt werden soll. Dort sind auch alle Punkte eingezeichnet, an die der Ursprung der anderen Testliste beim Verschieben angelegt wird. Von diesen ist für das Beispiel in jeder Liste einer besonders gekennzeichnet. Das ist in T_A der zweite Punkte (A_2) und in T_B der dritte Punkt (B_3). Die folgende Überlegung gilt jedoch für jedes Paar (A_i, B_j). Wird nun der Ursprung von T_B nach A_2 verschoben, so verschiebt sich auch B_3. Die neue Position von B_3 ist in der rechten Grafik als P_{A_2,B_3} eingezeichnet.

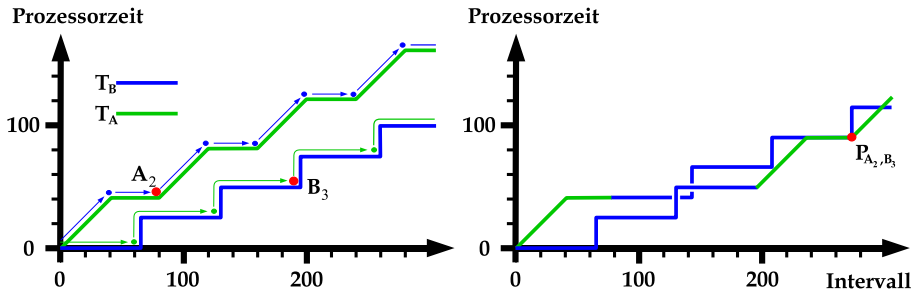


Abbildung 5.6: Der Punkt P_{A_2,B_3} wird sowohl beim Verschieben von T_A, als auch beim Verschieben von T_B erreicht.

Führt man dies umgekehrt durch und verschiebt den Ursprung von T_A nach B_3 so wird auch A_2 an die Position von P_{A_2,B_3} geschoben. Eine etwas formalere Begründung ergibt sich über die Vektoren $\overrightarrow{v_{A_2}} = (A_2.x, A_2.y)$ und $\overrightarrow{v_{B_3}} = (B_3.x, B_3.y)$. Verschiebt man nun den Ursprung von T_B nach A_2, so müssen hierzu die Vektoren aller Punkte von T_B mit $\overrightarrow{v_{A_2}}$ addiert werden, so dass sich P_{A_2,B_3} aus $\overrightarrow{v_{B_3}} + \overrightarrow{v_{A_2}}$ ergibt. Umgekehrt ergibt sich P_{A_2,B_3} dann beim Verschieben von T_A durch $\overrightarrow{v_{A_2}} + \overrightarrow{v_{B_3}}$.
Für die Komplexitätsuntersuchung könnte man nun annehmen, dass diese Punkte nicht zweimal in der Berechnung der maximalen Testlistenlänge auftreten. In der Praxis macht uns da leider die Gleitkommaarithmetik einen Strich durch die Rechnung. In den hier dargestellten Algorithmen wird P_{A_2,B_3} beim Verschieben von T_B im Prinzip ermittelt durch

$$P_{A_2,B_3} = \overrightarrow{v_{A_1}} + (\overrightarrow{v_{A_2}} - \overrightarrow{v_{A_1}}) + \overrightarrow{v_{B_1}} + (\overrightarrow{v_{B_2}} - \overrightarrow{v_{B_1}}) + (\overrightarrow{v_{B_3}} - \overrightarrow{v_{B_2}})$$

und beim Verschieben von T_A durch

$$P_{A_2,B_3} = \overrightarrow{v_{B_1}} + (\overrightarrow{v_{B_2}} - \overrightarrow{v_{B_1}}) + (\overrightarrow{v_{B_3}} - \overrightarrow{v_{B_2}}) + \overrightarrow{v_{A_1}} + (\overrightarrow{v_{A_2}} - \overrightarrow{v_{A_1}}).$$

Die Additionen werden auch in dieser Reihenfolge durchgeführt. Da für die Gleitkommaarithmetik jedoch nicht das Assoziativgesetz gilt [16], können die beiden Terme unterschiedliche Ergebnisse liefern. Daher muss in der Worst Case-Berechnung angenommen werden, dass jeder Punkt P_{A_i,B_j} durch zwei nah beieinander liegende Punkte dargestellt wird.
Allerdings lässt sich das Problem prinzipiell durch eine geschickte Implementierung lösen. In diesem Fall müsste sichergestellt werden, dass die Additionsschritte auf beiden Wegen in der selben Reihenfolge erfolgen. Dies ließe sich zum Beispiel realisieren, indem eine spezialisierte Minimum-Berechnung in der `MinimumSplit`-Funktion verwendet wird. Das Vorgehen müsste dann so sein, dass das Verschieben der Testlisten nicht direkt in der `MinimumSplit`-Funktion sondern während der Minimum-Berechnung stattfindet. Dazu muss bei der Minimumberechnung jedesmal bevor auf einen Punkt der zu verschiebenden Testliste zugegriffen wird, dieser Punkt um den gewünschten Wert verschoben werden. Dieses Vorgehen erhöht die Algorithmen-Komplexität nicht, da keine neuen Schleifeniterationen hinzukommen. Dafür wird jedoch erreicht, dass die Berechnung für jeden Punkt P_{A_i,B_j} jeweils auf beiden möglichen Wegen das gleiche Ergebnis liefert. Für das Beispiel von P_{A_2,B_3} gilt nun beim Verschieben der

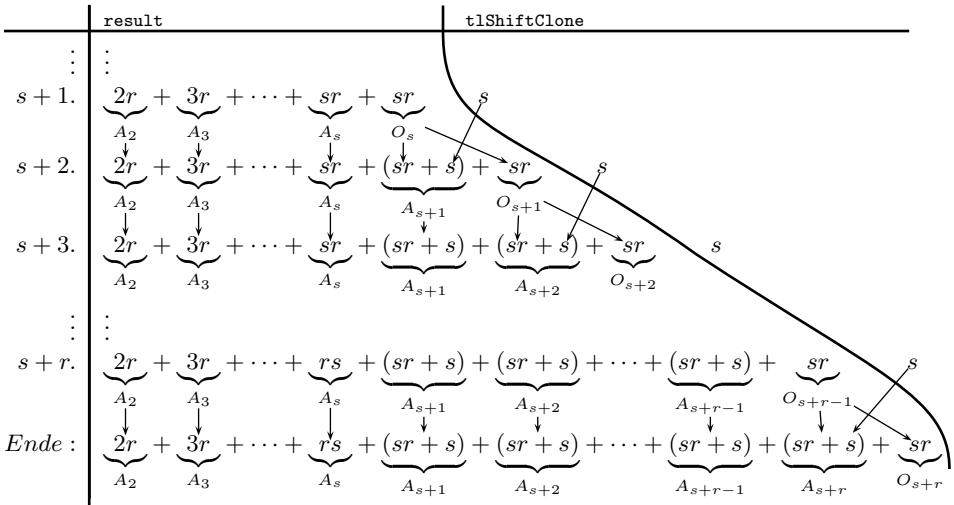
einen Liste

$$P_{A_2,B_3} = (\overrightarrow{v_{A_1}} + (\overrightarrow{v_{A_2}} - \overrightarrow{v_{A_1}})) + (\overrightarrow{v_{B_1}} + (\overrightarrow{v_{B_2}} - \overrightarrow{v_{B_1}}) + (\overrightarrow{v_{B_3}} - \overrightarrow{v_{B_2}}))$$

und beim Verschieben der anderen

$$P_{A_2,B_3} = (\overrightarrow{v_{B_1}} + (\overrightarrow{v_{B_2}} - \overrightarrow{v_{B_1}}) + ((\overrightarrow{v_{B_3}} - \overrightarrow{v_{B_2}})) + \overrightarrow{v_{A_1}} + (\overrightarrow{v_{A_2}} - \overrightarrow{v_{A_1}})).$$

Man beachte die Klammersetzung. Da die Addition in der Gleitkommaarithmetik kommutativ ist, liefern beide Wege dasselbe Ergebnis.
Eine solche Implementierung wird in dieser Arbeit nicht vorgenommen. Da sie jedoch machbar ist, soll für die weitere Komplexitätsbestimmung eine solche Umsetzung angenommen werden, um eine möglichst kleine Laufzeitabschätzung für die Algorithmen zu finden.
Diese angenommene Implementierung garantiert, dass jeder Punkt P_{A_i,B_j} im Ergebnis nur einmal vorkommt. Diese Eigenschaft kann nun in die Berechnung der Testlistenlänge einfließen.
In der Herleitung der Gleichung 5.2 wird für die ersten s Iterationen jeweils die gleiche Testliste mit r Elementen für die Minimum-Berechnung verwendet. Dabei werden bereits alle möglichen Punkte P_{A_i,B_j} erreicht. Das bedeutet, dass in den darauf folgenden r Iterationen keine weiteren Originalpunkte, sondern nur Schnittpunkte hinzukommen können. Mit dieser Überlegung bleibt die Berechnung der Testlistenlänge von `result` für die ersten s-Iterationen wie in Gleichung 5.2. Die nachfolgenden Iterationen ändern sich aber wie folgt:



$$\begin{aligned} &= \left(\sum_{i=2}^{s} i\right) r + r(sr+s) + sr \\ &= (\frac{s^2 r}{2} + \frac{sr}{2} - r) + r^2 s + 2sr \\ &= \frac{1}{2}s^2 r + r^2 s + \frac{5}{2}sr - r \end{aligned} \tag{5.3}$$

Im Vergleich zu Gleichung 5.2 zeigt sich jedoch, dass sich hierdurch nur die konstanten Faktoren ändern. Für die Komplexitätsuntersuchung sind diese allerdings nur von geringer Bedeutung (vgl. Abschnitt 5.1.2).

An dieser Stelle bleibt zudem offen, wie gut diese Abschätzung der Testlistenlänge ist. Es konnten insbesondere keine Testlisten als Operanden für die `MinimumSplit`-Funktion gefunden werden, die als Ergebnis eine Testliste mit einer Länge haben, die annähernd der hier gefunden Maximallänge entspricht und somit das Wachstum von s^2r erklärt. Ergebnisse mit einem ungefähren Wachstum von sr lassen sich dagegen relativ einfach finden.
Das bisherige Ergebnis ist also noch nicht zufrieden stellend. Mit der Vermutung, dass das Wachstum der Liste eigentlich durch sr begrenzt sein müsste, kann man in den vorherigen Gleichungen nun den Grund dafür suchen, warum die Wachstumsrate quadratisch von s abhängt. Dort erkennt man, dass der quadratische Faktor durch die Schnittpunkte entsteht, die in jeder Iteration berücksichtigt werden, nachdem sie einmal aufgetreten sind. Da es $s + r$ Iterationen sind, kann das ziemlich häufig sein. Allerdings ist es für den Algorithmus nicht notwendig, iterativ vorzugehen. Wie weiter oben bereits erwähnt, ist die Reihenfolge irrelevant, in der die einzelnen, verschobenen Testlisten für die Minimumfunktion verwendet werden. Im Speziellen ist es sogar möglich eine Divide & Conquer Strategie zu verfolgen. Die Testliste mit s Elementen muss beispielsweise r-mal verschoben und für die Minimumfunktion verwendet werden. Nun kann man jedoch auch diese r Berechnungen in zweimal $1/2r$ Berechnungen aufteilen und die Ergebnisse am Ende über die Minimumfunktion kombinieren. Diese zwei Teilaufgaben kann man nun immer weiter aufteilen, bis eine Aufgabe nur noch aus zwei Testlisten besteht. In Abbildung 5.7 ist das Vorgehen grafisch dargestellt für das Verschieben der Testliste mit n Elementen. Bei den nun folgenden Überlegungen muss nicht mehr unterschieden werden zwischen längerer und kürzerer Liste.

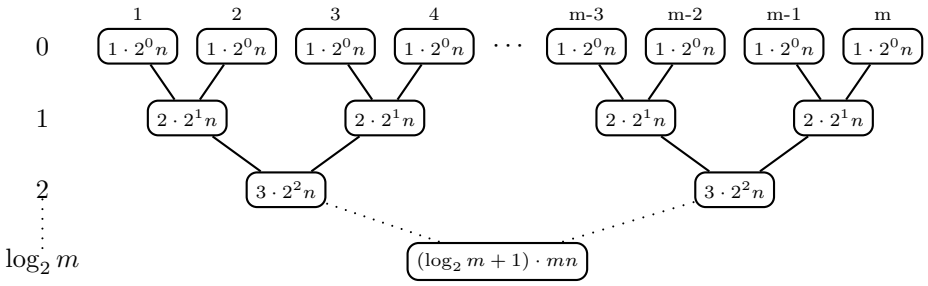
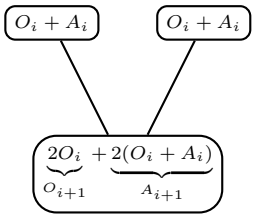


Abbildung 5.7: Divide & Conquer Strategie bei den Minimumberechnungen

Die Knoten im abgebildeten Baum enthalten die Längen der Testlisten. In der ersten Ebene sind dies die m verschobenen Testlisten der Länge n. Von je zwei dieser Listen wird das Minimum gebildet. Die maximalen Längen der sich dabei ergebenden Listen sind in den Knoten der nächsten Ebene eingetragen. Da es sich um einen Binärbaum mit m Blättern handelt, gibt es $\log_2 m$ Ebenen. Hier soll zuerst ein Spezialfall betrachtet werden, bei dem m eine 2-er Potenz ist, und es sich daher um einen vollständigen Binärbaum handelt. In einem solchen Baum besitzen alle Knoten einer Ebene den gleichen Wert.
Es soll jetzt gezeigt werden, wie sich die Werte der Knoten ergeben. Weiter oben wurde bereits dargestellt, dass bei hintereinander ausgeführten Minimumoperationen für die Ermittlung der maximalen Testlistenlänge zum Einen die Anzahl der Originalpunkte der Eingabelisten relevant sind und zum Anderen die Anzahl der Schnittpunkte, die bei vorherigen Minimumoperationen entstanden sind.

Da die Werte für alle Knoten einer Ebene gleich sind, bezeichnet im Folgenden O_i die Menge der Originalpunkte, die eine Liste in Ebene i haben kann und A_i dementsprechend die Anzahl der Schnittpunkte. Wird auf dieser Ebene nun eine Minimumoperation durchgeführt, so kann die Ergebnisliste alle Originalpunkte beider Eingabelisten beinhalten. Es gilt daher $O_{i+1} = 2 \cdot O_i$. Die Berechnungen beginnen mit den verschobenen Eingabelisten. Daher gilt $O_0 = n$. Daraus lässt sich einfach folgern:



$$O_i = 2^i n$$

Die Ergebnisliste kann im schlimmsten Fall als Schnittpunkte alle Schnittpunkte der Eingabelisten haben und es können durch die Originalpunkte neue Schnittpunkte entstanden sein. Daher ergibt sich die Anzahl der Schnittpunkte durch $A_{i+1} = 2O_i + 2A_i$. Über Induktion lässt sich nun zeigen, dass $A_i = i \cdot 2^i r$:

Induktionsanfang:	A_0	$=$	$0 = 0 \cdot 2^0 n$
Induktionsvoraussetzung:	A_i	$=$	$i \cdot 2^i n$
Induktionsschritt:	A_{i+1}	$=$	$2 \cdot O_i + 2 \cdot A_i$
		$=$	$2 \cdot 2^i n + 2 \cdot i \cdot 2^i n$
		$=$	$2^{i+1} n + i \cdot 2^{i+1} n$
		$=$	$(i+1) \cdot 2^{i+1} n$

Die Gesamtlänge ergibt sich nun aus der Summe der Schnitt- und Originalpunkte:

$$O_i + A_i = 2^i n + i \cdot 2^i n = (i+1) \cdot 2^i n \tag{5.4}$$

Für das Endergebnis ergibt sich also eine Länge von $(\log_2 m + 1) \cdot mn$. Dabei sind darin maximal $O_{\mathrm{TL}(n)} = mn$ Originalpunkte und $A_{\mathrm{TL}(n)} = \log_2 m \cdot mn$ Schnittpunkte enthalten. Bisher wurde jedoch nur die Liste mit n Elementen verschoben. Für die andere Liste ergibt sich auf dem gleichen Weg eine Liste der maximalen Länge $(\log_2 n + 1) \cdot mn$ mit $O_{\mathrm{TL}(m)} = mn$ und $A_{\mathrm{TL}(m)} = \log_2 n \cdot mn$. Zum Schluss muss noch das Minimum über diese beiden Listen gebildet werden. Wenn, wie weiter oben beschrieben, die Gesetzmäßigkeiten der Gleitkommaarithmetik bei der Implementierung beachtet wurden, sind die Originalpunkte beider Listen identisch. Daher besteht die Ergebnisliste maximal aus $O_{\mathrm{TL}(m)} = O_{\mathrm{TL}(n)} = mn$ Originalpunkten. Zusätzlich können noch die Schnittpunkte $A_{\mathrm{TL}(n)}$ und $A_{\mathrm{TL}(m)}$ enthalten sein, sowie noch einmal mn Schnittpunkte, die durch die Originalpunkte bei dieser letzten Minimumoperation entstehen können. Somit ergibt sich schließlich eine Testlistenlänge von $mn + mn + A_{\mathrm{TL}(n)} + A_{\mathrm{TL}(m)} = (\log_2 (mn) + 2) \cdot nm$. Das Wachstum einer solchen Funktion wird häufig als quasilinear in Abhängigkeit von mn bezeichnet [22].

Es wurde bisher gezeigt, dass diese Testlistenlänge nicht überschritten werden kann. Nun bleibt noch die Frage offen, wie gut diese obere Grenze ist. Das heißt, gibt es für beliebige Längen der Eingabelisten tatsächlich Listen, die ein solches Ergebnis bei der MinimumSplit-Operation erzeugen, oder kann sogar eine kleinere obere Grenze gefunden werden?

Gesucht ist nun eine Konstruktionsvorschrift, die beliebig lange Testlistenpaare erzeugt, für die das Ergebnis der MinimumSplit-Operation aus einer möglichst lange Testliste besteht, die im Idealfall die Länge $(\log_2 (mn) + 2) \cdot nm$ hat. In

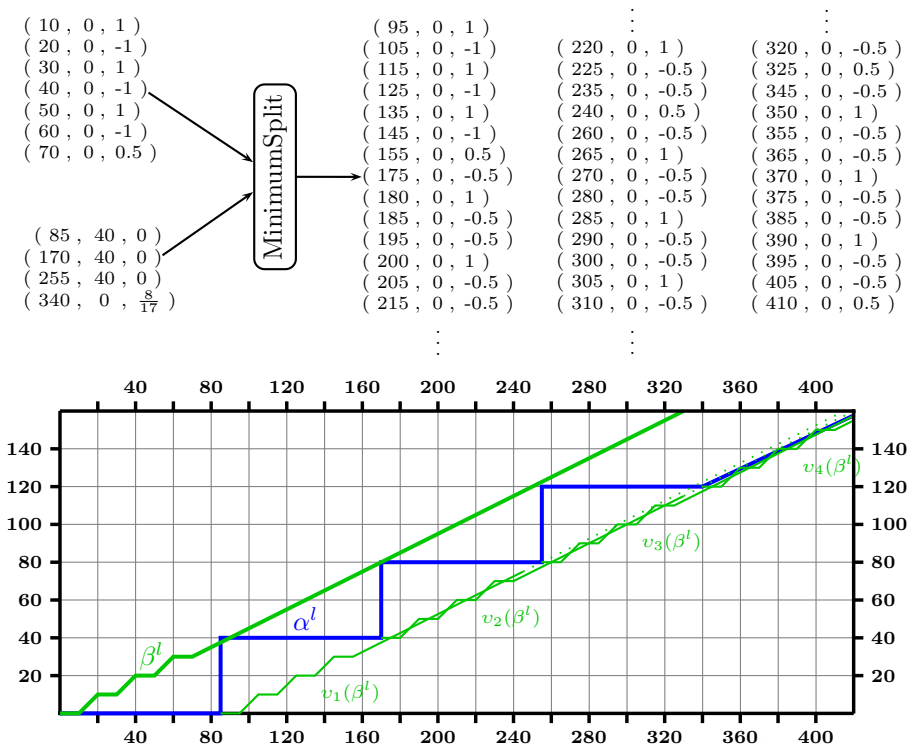


Abbildung 5.8: Erzeugung langer Testlisten durch die MinimumSplit-Operation

dieser Arbeit konnte jedoch lediglich eine gefunden werden, die Listen der Länge $mn+\frac{1}{2}(m-1)(n+1)-1$ erzeugt. Sie soll informell am Beispiel in Abbildung 5.8 erläutert werden. Hier wurde für die Länge der unteren Ankunftskurve $m = 4$ und für die untere Servicekurve $n = 7$ gewählt. Die Ankunftskurve wird so gebildet, dass sie eine periodische Taskaktivierung beschreibt, die in der m-ten Periode approximiert wird. Die Servicekurve wird dagegen so generiert, dass sich mit einer festen Periode jeweils Segmente mit Steigung 0 und 1 abwechseln. Nach $\frac{n-1}{2}$ Perioden, wird auch diese Kurve approximiert. Die Längen der Segmente werden nun so gewählt, dass beim Verschieben von β^l entlang von α^l das approximierte Segment von $\upsilon_i(\beta^l)$ alle Segmente von $\upsilon_{i+1}(\beta^l)$ schneidet, für alle $0 > i > m$. Beim Betrachten des Beispiels wird deutlich, dass die so erzeugte Testliste auf jeden Fall mehr als $m \cdot n$ Elemente enthält. Ohne hier näher darauf einzugehen, ergibt sich durch weitere, einfache Überlegungen eine Länge von $mn + \frac{1}{2}(m-1)(n+1) - 1$.
An dieser Stelle bleibt also für die Längen der Ergebnisses der MinimumSplit-Operation offen, ob sie in den schlimmsten Fällen linear oder quasilinear in Abhängigkeit von nm anwachsen.

Für die Laufzeitabschätzung müssen nun noch die Laufzeiten aller Minimumoperationen aufsummiert werden. Da die Laufzeit dieser Funktion für Testlisten der Längen k und l in $O(k+l)$ liegt, ergibt sich die Gesamtlaufzeit innerhalb der `MinimumSplit`-Funktion durch die Addition der Längen der Eingabelisten aller Minimumaufrufe. Da diese in den Knoten des Baums in Abbildung 5.7

eingetragen sind, müssen alle Knoten addiert werden:

$$
\begin{aligned}
& m \cdot 1 \cdot 2^0 n + \frac{m}{2} \cdot 2 \cdot 2^1 n + \frac{m}{4} \cdot 3 \cdot 2^2 n + \cdots + 1 \cdot (\log_2 m + 1) \cdot 2^{\log_2 m} n \\
= \; & \frac{m}{2^0} \cdot 1 \cdot 2^0 n + \frac{m}{2^1} \cdot 2 \cdot 2^1 n + \frac{m}{2^2} \cdot 3 \cdot 2^2 n + \cdots + \frac{m}{2^{\log_2 m}} \cdot (\log_2 m + 1) \cdot 2^{\log_2 m} n \\
= \; & 1 \cdot mn + 2 \cdot mn + 3 \cdot mn + \cdots + (\log_2 m + 1) \cdot mn \\
= \; & \sum_{i=1}^{\log_2 m + 1} i \cdot mn \\
= \; & \left(\frac{(\log_2 m + 1)^2}{2} + \frac{\log_2 m + 1}{2} \right) \cdot mn \\
= \; & \frac{1}{2} (\log_2 m)^2 \cdot mn + \frac{3}{2} \log_2 m \cdot mn + mn
\end{aligned} \tag{5.5}
$$

Dies ist das Ergebnis für den einen Baum, der die Minimumberechnungen für die Testliste mit n Elementen darstellt. Für die andere Liste müssen im Ergebnis m und n vertauscht werden. Die Summe beider Ergebnisse liefert schließlich die gesuchte Gesamtabschätzung:

$$
\frac{1}{2} \big((\log_2 m)^2 + (\log_2 n)^2 \big) \cdot mn + \frac{3}{2} \log_2 (nm) \cdot mn + 2mn \tag{5.6}
$$

Wie zu Beginn des Kapitels erwähnt, ist für die asymptotische Worst Case Laufzeit immer nur der Term mit dem schnellsten Wachstum für große Eingaben ohne konstante Faktoren relevant. Daher liegt die Gesamtlaufzeit des `min`-Aufrufs innerhalb der `MinimumSplit`-Funktion in $O(\big((\log_2 m)^2 + (\log_2 n)^2\big) \cdot mn)$. Da dies die größte Gesamtabschätzung für alle Anweisungen in `MinimumSplit` ist, liegt auch die Laufzeit von `MinimumSplit` in dieser Klasse.

5.3.12 SupSimple

Die Supremumoperation $\sup_{0 \leq t \leq \Delta} \{\epsilon(t)\}$, die zur Berechnung der Servicekurven benötigt wird, ist auf abstrahierten Testlisten eine sehr günstige Operation, da ihre Komplexität in O(n) liegt. Listing 5.9 zeigt, dass die Laufzeit linear mit der Länge der Testliste wächst, da der Algorithmus hauptsächlich aus einer `for`-Schleife (Zeile 10) besteht, deren Rumpf eine konstante WCET besitzt.
Der dort dargestellte Code stellt eine konkrete Implementierung des in [2] vorgestellten Algorithmus dar.
In den Zeilen 13, 28, 34, 42 und 58 wird die `addLast`-Operation verwendet. Da ansonsten keine Basisoperationen für dynamische Mengen verwendet werden, kann für diesen Algorithmus als Struktur der Testliste die einfach verkettete Liste gewählt werden. Dadurch kann `addLast` in konstanter Zeit ausgeführt werden.
Die `addLast`-Aufrufe sind zudem für die Länge der resultierenden Testliste verantwortlich. Der erste Aufruf in Zeile 13 kann in jedem Schleifendurchlauf aufgerufen werden, bis auf den ersten, da die initiale Belegung von `intersect` die umklammernden `if`-Anweisung blockiert. Die nächsten Aufrufe in Zeile 28, 34 und 42 schließen sich in jedem Durchlauf gegenseitig aus, was leicht an der Struktur der `if`-Anweisungen (Z. 26, 27, 40) abzulesen ist. Daher werden sie insgesamt maximal n mal aufgerufen. Der letzte Aufruf in Zeile 58 liegt dagegen

```
   S  T
1      public static TestList supSimple(TestList tl) {
2        double cResult = 0; // cost value, used for the resulting testlist
3        double gResult = 0; // gradient value, used for the resulting testlist
4        double intervalOld = 0; // marks the last considered interval length
5     1  double intersect = Double.POSITIVE_INFINITY; // next expected intersection
6        double cInput = 0; // cost value, used for the input testlist
7        double gInput = 0; // gradient value, used for the input testlist
8        TestList result = new TestList();

10    n  for (TestListElement tle : tl) {
11         // Handle intersections between previous and current element
12         if (tle.getOffset() > intersect) {
13   n-1●    result.addLast(new TestListElement(intersect, 0, gInput - gResult));
14           gResult = gInput;
15    n      cInput += (intersect - intervalOld) * gInput;
16           intervalOld = intersect;
17           intersect = Double.POSITIVE_INFINITY;
18         }

20         // update values for the current position
21         cInput += (tle.getOffset() - intervalOld) * gInput + tle.getCosts();
22         gInput += tle.getGradient();
23    n    cResult +=(tle.getOffset() - intervalOld) * gResult;
24         intervalOld = tle.getOffset();

26         if (cInput >= cResult) {
27           if (gInput > 0) {
28   ●         result.addLast(new TestListElement(tle.getOffset(),
29                                                 cInput - cResult,
30                                                 gInput - gResult));
31             gResult = gInput;
32             cResult = cInput;
33           } else {
34   ●         result.addLast(new TestListElement(tle.getOffset(),
35     n                                           cInput - cResult,
36                                                 - gResult));
37             gResult = 0;
38             cResult = cInput;
39           }
40         } else {
41    n      if (gResult > 0) {
42   ●         result.addLast(new TestListElement(tle.getOffset(), 0, -gResult));
43             gResult = 0;
44           }
45           if (gInput > gResult) {
46             intersect = tle.getOffset()
47                       + ((cInput - cResult)/(gResult - gInput));
48           } else {
49             if (intersect > tle.getOffset()) {
50               //previously calulated intersection point is useless
51               intersect = Double.POSITIVE_INFINITY;
52             }
53           }
54         }
55       }
56       // Handle intersections that occure after the last testlist element
57       if !Double.isInfinite(intersect) {
58   1●    result.addLast(new TestListElement(intersect, 0, gInput - gResult));

60     return result;
```

Listing 5.9: TestList.supSimple()

außerhalb der Schleife. Somit kann als Obergrenze für die Länge der resultierenden Testliste $n - 1 + n + 1 = 2n$ angegeben werden.
Betrachtet man nun den Kontext, in dem diese Funktion verwendet wird, so lässt sich diese Grenze noch etwas genauer angeben. Im abgebildeten Algorithmus sieht man, dass in einem Schleifendurchlauf nur dann zwei Testlisten-Element der resultierenden Liste hinzugefügt werden können, wenn `intersect` kleiner als ∞ ist (Z.12). Einen solchen Wert kann `intersect` jedoch nur in Zeile 46

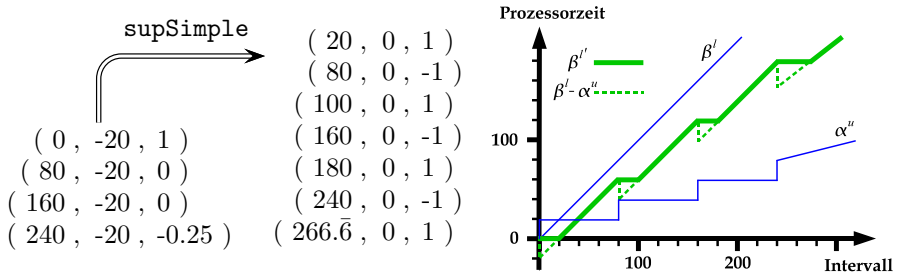


Abbildung 5.9: Servicekurvenberechnung für höchstpriorisiertesten Task

annehmen und dorthin gelangt der Algorithmus aufgrund der umklammernden `if`-Anweisungen (Z. 26,40,41) nur, wenn der aktuelle Wert der Eingabekurve unterhalb eines vorherigen, lokalen Maximums liegt. Das bedeutet, dass die Eingabekurve eine negative Steigung gehabt haben muss. Die `supSimple`-Funktion wird jedoch immer auf der Differenz der Service- und Ankunftskurven ausgeführt. Da die Servicekurve monoton steigend ist, kann eine negative Steigung in der Differenzkurve nur durch die Elemente der Ankunftskurve erzeugt werden. Daher kann für die Berechnung der ausgehenden Servicekurven eine genauere Grenze von $n + 2m$ Elementen angegeben werden, wobei n die Anzahl der Testlisten-Elemente der eingehenden Servicekurve ist und m die der eingehenden Ankunftskurve. In Abbildung 5.9 ist die Berechnung der unteren ausgehenden Servicekurve des höchstpriorisierten Task einer Verarbeitungseinheit dargestellt. Dort erschließt sich das oben beschriebene auf etwas intuitiverem Weg. Zudem lässt sich anhand der Grafik erahnen, dass bei einer Systemanalyse die Ergebnisse relativ häufig nah an dieser Obergrenze liegen können.

5.3.13 MaximumRequest

Wie in Abschnitt 4.5.3 beschrieben, kann die MaximumRequest-Operation berechnet werden, indem die Differenz der stehenden und der bewegten Liste jeweils an den Elementen dieser Listen bestimmt wird. Für die Elemente der stehenden Liste wird dies in der Implementierung dieser Arbeit durch die `maxReqPart1`-Funktion bewerkstelligt. Für die Elemente der bewegten Liste erfolgt dies durch die `maxReqPart2`-Funktion.
Der Aufbau beider Funktionen weist gewisse Analogien zur `MinimumSplit`-Funktion auf. Insbesondere für den für die Laufzeit kritischen Teil sind ähnliche Überlegungen notwendig. Die beiden Teilfunktionen werden daher hier nicht separat sondern zusammen betrachtet.
Die Vorgehensweise bei beiden Teilalgorithmen ist nahezu identisch. Um dies zu verdeutlichen, wurden die Algorithmen in Listing 5.10 und 5.11 in Bereiche unterteilt, die mit Buchstaben markiert sind und jeweils in beiden Algorithmen auftreten. Wenn im Folgenden nun von so einem Bereich gesprochen wird, gilt die Aussage jeweils für beide Algorithmen.
Der Bereich I initialisiert die Variablen und Datenstrukturen der Funktion. Es ist der einzige Bereich, für den sich die Worst Case Laufzeit in den Algorithmen unterscheidet. Das liegt daran, dass jeweils zwei Testlisten verwendet werden. Das

ist zum Einen die `result`-Liste, in der das Ergebnis aufgebaut wird, und zum Anderen die `tlM`- bzw. `mirroredS`-Liste, die immer die verschobene Eingabeliste enthält, und daher das Pendant zur `tlShiftClone`-Liste in der `MinimumSplit`-Funktion ist. Da diese Listen für `maxReqPart1` mit Kopien der bewegenden Liste initialisiert werden müssen, ist hier die Laufzeit von der Länge der bewegenden Liste abhängig. Ansonsten ist die Laufzeit für die Initialisierung konstant. Im Folgenden wird `result` als Ergebnisliste und die jeweils andere als Arbeitsliste bezeichnet.
Der Rest der Funktionen besteht aus einer `for`-Schleife (F), die m- bzw. s-mal ausgeführt wird. Innerhalb der Schleife gibt es den Bereich FR, dessen Anweisungen eine konstante WCET haben und in der Schleife daher eine Gesamtlaufzeit in $O(\mathrm{m})$ bzw. $O(\mathrm{s})$ haben. Sie dienen dazu, Werte für die `lift`- und `shift`-Operationen zu berechnen.
Den größten Unterschied in der Funktionsweise zur `MinimumSplit`-Funktion stellt der Bereich W dar. Für ihn gibt es bei der `MinimumSplit`-Funktion kein

```
   S       T
1            public static TestList maxReqPart1(TestList stand, TestList move) {
2          m   TestList tlM = (TestList)move.clone();
3          m   TestList result = (TestList)move.clone();
4              TestListElement lifter, tleM;
5     I        double costs[] = {0, 0};
6          1   double gradient[] = {0, 0};
7              double oldOffset[] = {0, 0};
8              double lift, shift, shiftSum = 0;

10         s   for (TestListElement tleS : stand) {

12    F
13         s     costs[S] += (tleS.getOffset() - oldOffset[S]) * gradient[S];
14           gradient[S] += tleS.getGradient();

15             // Remove all elements of the moving clone up to
16             // the current offset of the standing list.
17       s+m   while ((!tlM.isEmpty()) &&
18                    (tlM.first().getOffset() - shiftSum <= tleS.getOffset())) {
19             tleM = tlM.pollFirst();
20     W       costs[M] += tleM.getCosts()
21         m               + (tleM.getOffset()-shiftSum-oldOffset[M]) * gradient[M];
22             gradient[M] += tleM.getGradient();
23             oldOffset[M] = tleM.getOffset() - shiftSum;
24           }

26                 lift = costs[M] + (tleS.getOffset()-oldOffset[M])*gradient[M] - costs[S];
27    F            shift = oldOffset[S] - tleS.getOffset();
28         s       shiftSum += shift;
29                 costs[S] += tleS.getCosts();

31             // Shift moving list.
32    S  s·m   tlM.shift(shift);
33             // Lift moving list and insert current gradient.
34 +1● L  s   tlM.addFirst(new TestListElement(0, lift, gradient[M]));
35             // Get maximum of tlM and the result of the previous iteration.
36 *1● M  *2  result = TestList.max(result, tlM);
37             // Unlift moving list and remove current gradient
38 -1● L      tlM.addFirst(new TestListElement(0, -lift, -gradient[M]));
       s
40    F        oldOffset[S] = tleS.getOffset();
41           }
42           return result;
43         }
```

$*^1 = (\log_2 s + 1) \cdot sm$

$*^2 = (\log_2 s)^2 \cdot sm$

Listing 5.10: TestList.maxReqPart1

Gegenstück. Da bei der `MaximumRequest`-Funktion nicht in allen Fällen die vollständige Arbeitsliste benötigt wird, sorgt dieser Bereich dafür, dass die Arbeitsliste immer die gerade benötigte Länge hat. Dabei iteriert die `while`-Schleife insgesamt einmal über die andere Eingabeliste, über die die `for`-Schleife eben nicht iteriert. Daher wird der Schleifenrumpf m- bzw. s-mal ausgeführt und die Schleifenbedingung $m + s$-mal geprüft.
Das Verschieben der Arbeitsliste geschieht auch hier separat für die X-Richtung (S) und Y-Richtung (L). Die maximale Gesamtlaufzeit dieser Bereiche ergibt sich wie bei den entsprechenden Bereichen der `MinimumSplit`-Funktion.
Nun bleibt noch der Bereich M, der die kritischste Anweisung enthält. Hier wird das Maximum aus der Arbeits- und der Ergebnisliste gebildet. Es ist das Gegenstück zur Minimumberechnung in der `MinimumSplit`-Funktion. Die Überlegung zur Gesamtlaufzeit sind daher auch ähnlich.
Der größte Unterschied zwischen den Maximum Request Funktionen und der Minimum Split Funktion besteht im Bereich W, da durch diesen nicht immer die vollständige Arbeitsliste für die Maximumfunktion benötigt wird. Für die Worst Case Laufzeit muss jedoch der schlimmste Fall betrachtet werden, bei dem in jeder Iteration die fast vollständig Arbeitsliste für die Maximum-Funktion verwendet wird. Damit dieser für beide Teilfunktionen eintritt, müssen möglichst viele Testlisten-Elemente der bewegenden Liste einen größeren Offset haben als das letzte Element der stehenden Liste.
Daher ist im schlimmsten Fall die Arbeitsliste so lang wie die entsprechende Eingabeliste. Dies ist bei der Minimum Split Funktion immer der Fall. Zudem lassen sich für die maximale Länge der Ergebnisliste bei der Maximumberechnung die gleichen Überlegungen anstellen, wie dies bei der Minimumberechung in der Minimum Split Funktion geschehen ist. Daher kann für die Gesamtlaufzeit der Maximumberechnung in beiden Teilfunktionen die Gleichung 5.6 verwendet werden.
Bei den Überlegungen zur MinimumSplit Funktion wurde gezeigt, dass jeder Originalpunkt, der beim Verschieben der einen Liste auftritt, auch beim Verschieben der anderen erscheint. Da diese Überlegung auch bei der Maximum Request Funktion zutrifft, ergibt sich für sie auch eine maximale Länge der Ergebnisliste von $(\log_2 ms + 3) \cdot sm$.

```
 S      T
1          public static TestList maxReqPart2(TestList stand, TestList move) {
2            TestList mirroredS = new TestList();
3            TestList result = new TestList();

5            TestListElement nextS = null;
6            Iterator<TestListElement> iterS = stand.iterator();
7            if (iterS.hasNext()) {
8              nextS = iterS.next();
9              if (nextS.getOffset() == 0) {
10   I{  1       mirroredS.addFirst(new TestListElement(0, 0, -nextS.getGradient()));
11             }
12           }

14           double costs[] = {0, 0};
15           double gradient[] = {0, 0};
16           double oldOffset[] = {0, 0};
17           double shift, lift, tmpGradient = 0;
18           boolean additional = false;

20       m   for (TestListElement tleM : tlMoving) {
21             costs[M] += tleM.getCosts();
22             costs[M] += (tleM.getOffset() - oldOffset[M]) * gradient[M];
23   F{        gradient[M] += tleM.getGradient();
       m
25             //shift the mirror list
26             shift = tleM.getOffset() - oldOffset[M];
27   S{m·s     mirroredS.shift(shift);
28   F{        if (additional)
       m         mirroredS.addFirst(new TestListElement(shift, 0, -gradient[S]));
30             additional = true;

32       m+s   while ((nextS != null) && (nextS.getOffset() < tleM.getOffset())){
33               // calculate current costs etc.
34               costs[S] += nextS.getCosts()
35               costs[S] += (nextS.getOffset() - oldOffset[S]) * gradient[S];
36               gradient[S] += nextS.getGradient();
37               oldOffset[S] = nextS.getOffset();
38               // add mirrored element
39   W{  s       mirroredS.addFirst(new TestListElement(tleM.getOffset()-nextS.getOffset(),
40                                                      nextS.getCosts(),
41                                                      -nextS.getGradient()));
42               // get next element if any
43               if (nextS.getOffset() == tleM.getOffset()) additional = false;
44               nextS = iterS.hasNext() ? iterS.next() : null;
45             }

47             if (additional)
48               mirroredS.addFirst(new TestListElement(0, 0, gradient[S]));
50   F{      m   //Lift mirror list.
51             lift = costs[M] - costs[S]
52                    - (tleM.getOffset() - oldOffset[S]) * gradient[S];
53 +1● L{       mirroredS.lift(lift);
54             //Maximum of the mirrored list and the previous maximum
55 *1● M{  *2   result = TestList.max(result, mirroredS);
56             //unlift mirror list
57 -1● L{       mirroredS.lift(-lift);
       m
59   F{        oldOffset[M] = tleM.getOffset();
60           }
61       1   return result;
62         }
```

$*^1 = (\log_2 m + 1) \cdot sm$

$*^2 = (\log_2 m)^2 \cdot sm$

Listing 5.11: TestList.maxReqPart2

5.3.14 Zusammenfassung

Nun sind alle Teilfunktionen untersucht, die für die Berechnung der ausgehenden Kurven benötigt werden. Zu Beginn des Kapitels wurde kurz beschrieben, dass die Wahl der Datenstrukturen Einfluss auf die Laufzeiten der Algorithmen haben kann. Dort wurden als Kandidaten für die Testlisten verkettete Listen und Red-Black-Trees vorgeschlagen, die je nach Anwendung unterschiedliche Vor- und Nachteile haben.
In allen hier untersuchten Algorithmen wurden jedoch ausschließlich Operationen auf den Testlisten verwendet, die am effektivsten mit einer verketteten Liste als Datenstruktur ausgeführt werden können. Damit zeigt sich, dass die anfängliche Wahl des Red-Black-Trees als Struktur nicht optimal ist.

Um an dieser Stelle noch einmal einen Überblick über die Ergebnisse bei den einzelnen Operationen zu geben, sind hier die ermittelten Werte für das asymptotische Laufzeitverhalten und die Länge der Testlisten aufgelistet.

Funktion	asymptotische Laufzeit	maximale Länge der Ergebnisliste
Traversieren	n	unverändert
Kopieren	n	n
Vereinigung	$n+m$	$n+m$
Negation	n	unverändert
Skalierung	n	unverändert
Shift	n	unverändert
Lift	1	$n+1$
Subtraktion	$n+m$	$n+m$
Minimum	$n+m$	$2(n+m)$
Maximum	$n+m$	$2(n+m)$
SupSimple	n	$2n$
MinimumSplit	$\left((\log_2 m)^2 + (\log_2 n)^2\right) \cdot mn$	$\left(\log_2 (mn) + 2\right) \cdot nm$
MaximumRequest	$\left((\log_2 m)^2 + (\log_2 n)^2\right) \cdot mn$	$\left(\log_2 (mn) + 2\right) \cdot nm$

Tabelle 5.1: Laufzeiten und Ergebnisgrößen der Teilfunktionen

5.4 Tasktransformation

In Abbildung 3.6 wurde bereits eine vollständige Tasktransformation mit den vor- und nachbereitenden Operationen dargestellt. Dort werden die Ankunftskurven vor dem Betreten des Taskknotens mit der Ausführungszeit des Tasks und nach dem Verlassen mit dem Kehrwert der Ausführungszeit skaliert. Betrachtet man dies auf der Systemebene, so muss jede Ankunftskurve, die von einem Task zu einem anderen führt, zweimal skaliert werden. Kombiniert man jedoch diese Skalierungen, wie es in Abbildung 5.10 zu sehen ist, so spart man sich für jeden Taskknoten eine Skalierungsoperation.
Es muss daher für jeden Taskknoten die Skalierung aller Ankunftskurven, deren Addition und schließlich die Berechnung der ausgehenden Ankunfts- und Ser-

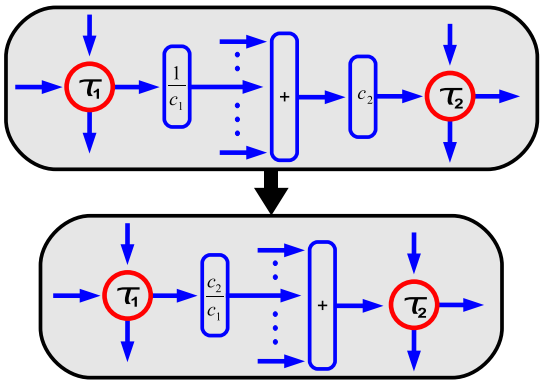


Abbildung 5.10: Reduzierung der Skalierungsoperationen

vicekurven durchgeführt werden. Alle beteiligten Kurven bestehen dabei aus je zwei Testlisten, die die oberen und unteren Grenzen darstellen. Für die Komplexitätsanalyse müssen nun für alle beteiligten Testlisten die maximalen Längen ermittelt werden, um daraus die maximale Laufzeit der einzelnen Berechnungen zu bestimmen. In Abbildung 5.11 sind sämtliche Operationen im Zusammenhang zu sehen. Dabei sind alle Kurven in obere und untere Grenze unterteilt, die mit den maximal möglichen Längen beschriftet sind. Die Längen der eingehenden unteren Ankunftskurven sind dabei a_1^l bis a_n^l, die der oberen Ankunftskurven a_1^u bis a_n^u und die der unteren und oberen Servicekurve b^l und b^u.
Die Herleitung der Testlistenlängen für die ausgehenden Kurven erfolgt in den folgenden Unterabschnitten. Zudem wird die Laufzeit der einzelnen Operationen und schließlich der gesamten Tasktransformation ermittelt.

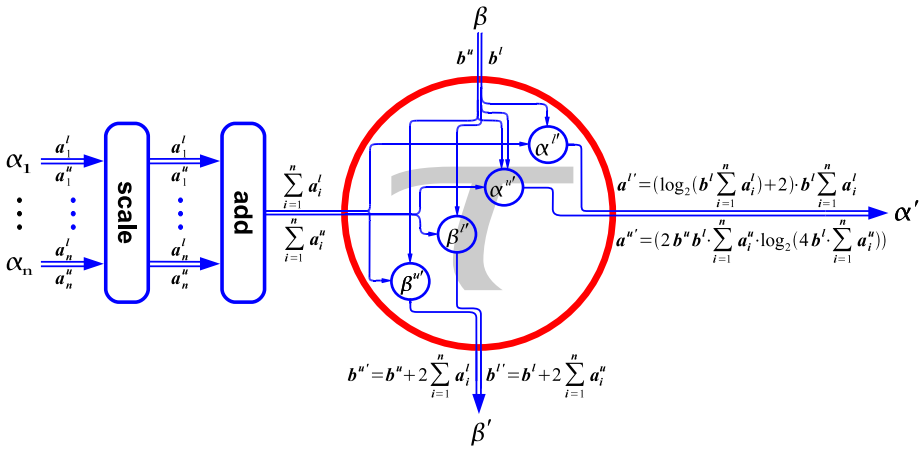


Abbildung 5.11: Die Operationen einer Tasktransformation

5.4.1 Vorbereitende Operationen

Die Laufzeitkomplexität der Skalierung zu bestimmen ist relativ einfach, da alle eingehenden Ankunftskurven unabhängig voneinander skaliert werden. Daher ergibt sich die Gesamtlaufzeitabschätzung aus der Summe der Einzelabschätzungen. Mit dem Ergebnis aus Abschnitt 5.3.5 ist dies $O(\sum_{i=1}^{n}(a_i^l + a_i^u))$. Die Längen der Testlisten ändern sich dabei nicht.

Bei der Addition ist dies etwas schwieriger. In Abschnitt 5.3.3 wurde die Addition zweier Testlisten vorgestellt. Hier sollen jetzt jedoch n Testlisten addiert werden. Um dies zu realisieren, kann man die Addition zweier Testlisten als Primitive benutzen und daraus die Addition von n Testlisten aufbauen. Ein anderer möglicher Weg ist, die Addition zu generalisieren und eine neue Funktion zu entwickeln, die direkt die Vereinigung von n Listen durchführt.
Es ist davon auszugehen, dass in vielen Fällen die Anzahl der Quellen, die einen Task aktivieren können, vergleichsweise gering ist, und die Art der Implementierung der Addition daher für die Laufzeit der Gesamtanalyse relativ unbedeutend ist. Der Vollständigkeit halber wird auch die Laufzeit dieser Operation nun näher untersucht. In der Implementierung dieser Arbeit wurde ein recht einfacher Algorithmus verwendet, der die binäre Addition als Primitive verwendet. Daher werden die Möglichkeiten dieses Ansatzes nun vertieft betrachtet.

Binäre Addition als Primitive
Bei der Aufsummierung mehrerer Zahlen macht man sich normalerweise keine Gedanken darüber, in welcher Reihenfolge dies geschieht, da es meistens keine Auswirkungen auf das Ergebnis oder die Laufzeit hat. Nun darf man allerdings nicht den Fehler machen und dies auch bei den Testlisten voraussetzen. Hier kann die Laufzeit sehr stark mit der Reihenfolge der Additionen variieren. Das soll an einem kleinen Beispiel gezeigt werden.

```
1  public TestList addN(TestList[] testlists) {
2    TestList result = new TestList();
3    for (TestList tl : testlists) {
4      result.addTestList(tl);
5    }
6    return result;
7  }
```

Listing 5.12: Naive Implementierung der Addition

Wie in Abschnitt 5.3.3 erwähnt, liegt die Laufzeit der Addition zweier Testlisten der Längen a_i und a_j in $O(a_i + a_j)$. Eine naive Implementierung der Addition von n Testlisten könnte nun wie in Listing 5.12 aussehen.
Für diese Implementierung ist in der nebenstehenden Tabelle aufgelistet, wie die Laufzeitabschätzung für die einzelnen Iterationen bei der Addition der unteren Ankunftskurven ist. Das geschieht auf Basis der Laufzeitabschätzung der Addition zweier Testlisten. Zudem ist eingetragen, wie sich

Iteration	Laufzeit	Länge von `result`
1	$a_1^l + a_2^l$	$a_1^l + a_2^l$
2	$a_1^l + a_2^l + a_3^l$	$a_1^l + a_2^l + a_3^l$
3	$\sum_{i=1}^{4} a_i^l$	$\sum_{i=1}^{4} a_i^l$
$\vdots$	$\vdots$	$\vdots$
n-1	$\sum_{i=1}^{n} a_i^l$	$\sum_{i=1}^{n} a_i^l$

die Länge der `result`-Liste dabei entwickelt.
Für die Gesamtlaufzeit muss nun einfach die Laufzeit aller Iterationen aufsummiert werden:

$$(n-1)\cdot a_1^l + (n-1)\cdot a_2^l + (n-2)\cdot a_3^l + \ldots + 1\cdot a_n^l = \sum_{i=1}^{n}\big((n-i)\cdot a_i^l\big) - a_1^l \quad (5.7)$$

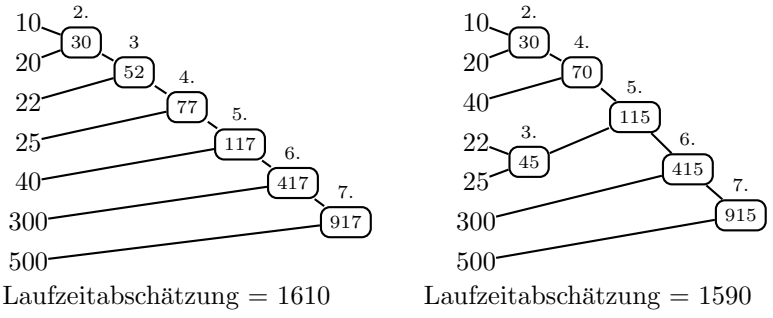
Nun kann man sich als Beispiel einen Task mit sieben eingehenden Ankunftskurven vorstellen, deren Testlisten für die untere Grenze die Längen 10, 20, 22, 25, 40, 300 und 500 haben.
Führt man die Additionen nun aufsteigend sortiert nach den Längen der Listen durch, so ergibt der obige Term einen Wert von

$$1610 = 6\cdot 10 + 6\cdot 20 + 5\cdot 22 + 4\cdot 25 + 3\cdot 40 + 2\cdot 300 + 1\cdot 500.$$

Macht man es jedoch mit absteigender Sortierung erhält man

$$5216 = 6\cdot 500 + 6\cdot 300 + 5\cdot 40 + 4\cdot 25 + 3\cdot 22 + 2\cdot 20 + 1\cdot 10.$$

Der Grund für den höheren Wert im zweiten Fall liegt darin, dass die Elemente der längsten Liste gleich zu Beginn in die `result`-Liste einfließen und danach bei jeder Iteration beachtet werden müssen. Es sollten also immer die kürzesten Listen zuerst addiert werden, um den günstigsten Fall zu erhalten. Das gilt auch für Zwischenergebnisse. Der Wert von 1610 lässt sich für dieses Beispiel nämlich noch weiter reduzieren. In den umrahmten Knoten des Baumes in Abbildung 5.12(a) ist die Länge der `result`-Liste nach jeder Addition eingetragen. Die Kinder eines solchen Knotens enthalten immer die Längen der Eingabelisten. Die dort abgebildete Struktur ergibt sich, da die Testlisten in der Reihenfolge ihrer Längen, nacheinander addiert werden. Nun kann man jedoch die `result`-Liste wie alle anderen Listen behandeln und sie dynamisch nach jeder Addition in die Reihe der übrigen Listen einsortieren. Addiert man jetzt immer die zwei kürzesten Listen, so ergibt sich der Baum in Abbildung 5.12(b). Die Laufzeit-



(a) statische Sortierung

(b) dynamische Sortierung mit der `result`-Liste

Abbildung 5.12: Varianten der Addition

abschätzung ergibt für diesen Fall einen Wert von 1590. Er lässt sich ermitteln, indem die Werte der umrandeten Knoten des Baumes addiert werden. Der Effekt der dynamischen Sortierung ist hier gering. Er erhöht sich jedoch, je mehr

sich die Längen der Listen gleichen.
Um nun aber zu einer generellen Laufzeitabschätzung zu kommen, soll hier betrachtet werden, was passiert, wenn man das Problem mit der Divide & Conquer Strategie löst und immer zwei Testlisten nimmt und addiert. Aus diesen Ergebnissen nimmt man wieder je zwei und addiert sie, usw. Das ist im folgenden Baum dargestellt. In jeder Ebene eines solchen Baumes ist die Summe der Knoten kleiner oder gleich $\sum_{i=1}^{n} a_i$. Da sich die Laufzeitabschätzung einer Addition aus der maximalen Länge des Ergebnisses ergibt, ist dies zugleich auch die Laufzeitabschätzung für alle Additionen in dieser Ebene. Für die Gesamtlaufzeit können nun die Laufzeiten aller $\log_2 n$ Ebenen addiert werden. Somit liegt die Laufzeit der Divide & Conquer Strategie in $O(\log_2 n \cdot \sum_{i=1}^{n} a_i)$.

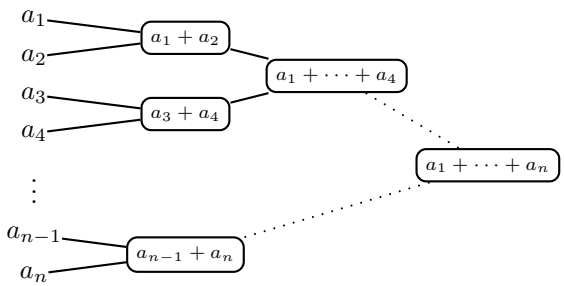


Abbildung 5.13: Divide & Conquer Strategie für die Addition von n Testlisten

Die Variante des dynamischen Sortieren der Testlisten wird in den meisten Fällen wahrscheinlich eine bessere Laufzeit erzielen als die Divide & Conquer Variante. Da für sie jedoch hier keine bessere Laufzeitkomplexität angegeben werden kann, wird hier die Laufzeitabschätzung der Divide & Conquer-Strategie verwendet.

5.4.2 Servicekurven

Da die Komplexitätsklassen für die Supremumbildung und die Subtraktion zweier Testlisten nun bekannt sind, kann jetzt auch für die Berechnung der ausgehenden Servicekurve die Komplexität bestimmt werden. In Listing 5.13 ist sehr einfach zu erkennen, dass diese in O(n+m) liegt, da die Operationen lediglich sequentiell verknüpft werden müssen. Dabei ist n die Länge der Testliste der Servicekurve und m die der Ankunftskurve. Da das Vorgehen bei beiden Schranken gleich ist, gilt dies für die untere und obere Servicekurve.
Die Länge der Testliste nach der `supSimple`-Funktion ist im allgemeinen Fall maximal doppelt so lang wie die Eingabeliste, die hier eine maximale Länge von $n + m$ hat. In Abschnitt 4.5.1.2 wurde jedoch gezeigt, dass die Länge im hier betrachteten Kontext auf $n + 2m$ beschränkt ist. Dies ist daher die Länge der ausgehenden Servicekurve.

```
     S     T
1            public void calcService(TestList arrivalIn, TestList serviceIn) {
2              Testliste difference;
3  n+m   n+m   difference = serviceIn.subTestList(arrival);
4 n+2m   n+m   return TestList.supSimple(difference);
5            }
```

Listing 5.13: TestList.calcService

5.4.3 Ankunftskurven

Die untere Ankunftskurve wird allein über die `minimumSplit`-Funktion berechnet. Daher entspricht der Zeitaufwand und die Länge der resultierenden Testliste den in Abschnitt 5.3.11 gezeigten Grenzen.

```
1  public TestList calcUpperArrival(TestList upperArrival, ServiceCurve sc) {
2    // maximum request
3    TestList tmp = TestList.maximumRequest(sc.getLowerBound(), upperArrival);

5    // minimum split
6    return TestList.minimumSplit(sc.getUpperBound(), tmp);
7  }
```

Listing 5.14: TestList.calcUpperArrival

Für die obere Ankunftskurve muss dagegen sowohl die `MinimumSplit`-, als auch die `MaximumRequest`-Funktion ausgeführt werden. Sei m die Länge der oberen Ankunftskurve, n die Länge der oberen Servicekurve und s die Länge der unteren Servicekurve. Dann liegt die Laufzeit der Maximum Request Anweisung in $O(\big((\log_2 m)^2 + (\log_2 s)^2\big) \cdot ms)$ und erzeugt eine Liste mit nicht mehr als $(\log_2 (ms) + 2) \cdot ms$ Elementen. (vgl. Abschnitt 5.3.13)
Diese Liste ist nun wiederum ein Parameter der `minimumSplit`-Anweisung. Daher ergibt sich hier eine Laufzeit von

$$
\begin{aligned}
& \left(\left(\log_2\Big(\big(\log_2(ms)+2\big)\cdot ms\Big)\right)^2 + (\log_2 n)^2\right) \cdot \big(\log_2(ms)+2\big)\cdot msn \\
= \; & \left(\left(\log_2\Big(\big(\log_2(ms)+\log_2 4\big)\cdot ms\Big)\right)^2 + (\log_2 n)^2\right) \cdot \big(\log_2(ms)+2\big)\cdot msn \\
= \; & \left(\left(\log_2\big(\log_2(4ms)\cdot ms\big)\right)^2 + (\log_2 n)^2\right) \cdot \log_2(4ms)\cdot msn \\
= \; & \left(\left(\log_2\big(\log_2(4ms)\big)+\log_2(ms)\right)^2 + (\log_2 n)^2\right) \cdot \log_2(4ms)\cdot msn \\
= \; & \left(\left(\log_2\big(\log_2(4ms)\big)+\log_2(ms)\right)^2 + (\log_2 n)^2\right) \cdot \log_2(4ms)\cdot msn \\
= \; & \Big(\underbrace{\Big(\log_2\big(\log_2(4ms)\big)\Big)^2}_{A} + \underbrace{2\cdot\log_2\big(\log_2(4ms)\big)\cdot\log_2(ms)}_{B} + \underbrace{\big(\log_2(ms)\big)^2}_{C} + \underbrace{(\log_2 n)^2}_{D}\Big) \\
& \cdot \log_2(4ms)\cdot msn
\end{aligned}
$$

Da auf lange Sicht das Wachstum von $\log_2(ms)$ immer größer ist als das von $\log_2\big(\log_2(4ms)\big)$, ist dementsprechend auch der Summand C in der obigen Gleichung immer größer als A und B. Bei der Einordnung der Laufzeitkomplexität ist nach dem Wachstum der Laufzeit für große Eingaben gefragt. Somit können die Summanden A und B hier ignoriert werden. Daher ergibt sich für die Berechnung der oberen Ankunftskurve eine Laufzeitkomplexität von

$$\Big(\big(\log_2(ms)\big)^2 + (\log_2 n)^2\Big) \cdot \log_2(ms)\cdot msn \tag{5.8}$$

Auch die maximale Länge der ausgehenden oberen Ankunftskurve kann mit den

Formeln aus den vorherigen Abschnitten bestimmt werden:

$$\begin{aligned} & \Big(\log_2\Big(\big(\log_2(ms)+2\big)\cdot msn\Big)+2\Big)\cdot\big(log_2(ms)+2\big)\cdot msn \\ = \; & \Big(\log_2\big(\log_2(4ms)\cdot msn\big)+3\Big)\cdot log_2(4ms)\cdot msn \\ = \; & \log_2\big(4msn\cdot\log_2(4ms)\big)\cdot log_2(4ms)\cdot msn \end{aligned} \tag{5.9}$$

Diese Formel ist relativ unübersichtlich. Da jedoch eine obere Grenze für die Länge der Testliste gesucht ist, kann auch eine etwas größere obere Grenze angegeben werden, die dafür einfacher darzustellen ist. Für alle $m, n, s \geq 1$ gilt sowohl $4msn \geq 4ms$, als auch $4msn > \log_2(4ms)$. Daher ist auch folgende Formel korrekt:

$$\begin{aligned} = \; & \log_2\big(4msn\cdot\log_2(4ms)\big)\cdot log_2(4ms)\cdot msn \\ < \; & \log_2(4msn\cdot 4msn)\cdot log_2(4msn)\cdot msn \\ = \; & 2\cdot\log_2(4msn)\cdot log_2(4msn)\cdot msn \\ = \; & 2msn\cdot\big(\log_2(4msn)\big)^2 \end{aligned} \tag{5.10}$$

Für die Komplexitätsanalyse einer vollständigen Systemanalyse mit dem Real-time Calculus bietet es sich nun an, diese etwas einfachere Formel zu verwenden.

5.4.4 Gesamtlaufzeit

In der folgenden Tabelle sind die Komplexitätsklassen aller Operationen einer Tasktransformation aufgelistet.

Funktion	O
Addition	$f_1 = \log_2 n \cdot \sum_{i=1}^{n} (a_i^l + a_i^u)$
Skalierung	$f_2 = \sum_{i=1}^{n} (a_i^l + a_i^u)$
untere Servicekurve	$f_3 = b^l + \sum_{i=1}^{n} a_i^u$
obere Servicekurve	$f_4 = b^u + \sum_{i=1}^{n} a_i^l$
untere Ankunftskurve	$f_5 = \big((\log_2 \sum_{i=1}^{n} a_i^l)^2 + (\log_2 b^l)^2\big) \cdot b^l \sum_{i=1}^{n} a_i^l$
obere Ankunftskurve	$f_6 = \Big(\big(\log_2 b^l\big)^2 + \Big(\log_2\big(\sum_{i=1}^{n} a_i^u \cdot b^u\big)\Big)^2\Big) \cdot \log_2\big(\sum_{i=1}^{n} a_i^u \cdot b^u\big) \cdot \sum_{i=1}^{n} a_i^u \cdot b^u \cdot b^l$

Tabelle 5.2: Komplexitätsklassen der an einer Tasktransformation beteiligten Operationen

Um die asymptotische Laufzeit der gesamten Transformation zu erhalten, müssen die einzelnen Laufzeitabschätzungen addiert werden. Dabei können die Summanden ignoriert werden, die für steigende Variablenwerte in jedem Fall ein kleineres Wachstum aufweisen als die verbleibenden Summanden (vgl. Abschnitt 5.1.2).

Das gilt für f_1, f_2, f_3 und f_4, da $f_1 \leq f_5 + f_6$, $f_2 \leq f_5 + f_6$, $f_3 \leq f_6$ und $f_4 \leq f_5 + f_6$. Für f_1 ist der Fall nicht gleich offensichtlich, weil der Faktor $\log_2 n$, so in keinem anderen Term vorkommt. Da n jedoch die Anzahl der eingehenden Servicekurven ist, gilt $n \leq \sum_{i=1}^{n} a_i^l$ und $n \leq \sum_{i=1}^{n} a_i^u$. Somit gilt weiter:

$$\begin{aligned} f_1 &= \log_2 n \cdot \sum_{i=1}^{n} (a_i^l + a_i^u) \\ &\leq \log_2 \left(\sum_{i=1}^{n} a_i^l\right) \cdot \sum_{i=1}^{n} (a_i^l) + \log_2 \left(\sum_{i=1}^{n} a_i^u\right) \cdot \sum_{i=1}^{n} (a_i^u) \\ &\leq f_5 + f_6. \end{aligned} \tag{5.11}$$

Die Komplexität einer Tasktransformation liegt daher in $O(f_5 + f_6)$. Fasst man $\sum_{i=1}^{n} a_i^l$ zu a^l und $\sum_{i=1}^{n} a_i^u$ zu a^u zusammen so ist dies:

$$O\Big(\big((\log_2 a^l)^2 + (\log_2 b^l)^2\big) \cdot b^l a^l + \Big((\log_2 b^l)^2 + \big(\log_2 (a^u b^u)\big)^2\Big) \cdot \log_2 (a^u b^u) \cdot a^u b^u b^l\Big) \tag{5.12}$$

Diese obere Grenze für die Laufzeit wird später auch für die Komplexitätsuntersuchung bei der Systemanalyse verwendet. Um das grundsätzliche Wachstum dieser Funktion etwas intuitiver einordnen zu können, kann jedoch eine größere, aber übersichtlichere obere Grenze angegeben werden durch:

$$O(f_5 + f_6) \quad \subseteq \quad O\Big(\big(\log_2 (b^l b^u a^l a^u)\big)^2 \cdot b^l b^u a^l a^u\Big). \tag{5.13}$$

Für die Komplexitätsanalyse einer vollständigen Systemanalyse sind auch wieder die Längen der Ergebnislisten interessant. Sie sind in der folgenden Tabelle noch einmal zusammengefasst.

Kurve	maximale Anzahl an Elementen
untere Servicekurve	$b^l + 2\sum_{i=1}^{n} a_i^u$
obere Servicekurve	$b^u + 2\sum_{i=1}^{n} a_i^l$
untere Ankunftskurve	$\big(\log_2 (b^l \cdot \sum_{i=1}^{n} a_i^l) + 2\big) \cdot b^l \cdot \sum_{i=1}^{n} a_i^l$
obere Ankunftskurve	$2b^l b^u \cdot \sum_{i=1}^{n} a_i^u \cdot log_2(4b^l b^u \cdot \sum_{i=1}^{n} a_i^u)$

Tabelle 5.3: Maximale Längen der ausgehenden Testlisten

5.5 Systemanalyse

5.5.1 Worst Case Szenario

Bei den bisherigen Untersuchungen zur Komplexität konnte die Laufzeit der Algorithmen oder die Länge der Ergebnisliste für einfache Funktionen meistens direkt aus dem Quellcode abgeleitet werden. Bei aufwendigeren Operationen, wie der MinimumSplit-Funktion, musste zudem noch die Vorgehensweise der Algorithmen tiefer durchleuchtet werden.

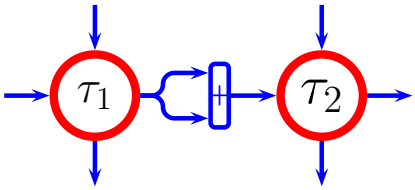


Abbildung 5.14: Ausgeschlossene Verknüpfung zweier Tasks

Bei der Komplexitätsbestimmung einer vollständigen Systemanalyse treten jedoch neue Probleme auf. Bisher wurden die Worst Case Laufzeiten der Algorithmen durch die O-Notation in Klassen eingeteilt. Der Wert, der dabei angegebenen Funktion, ist immer abhängig von den Größen, die die Laufzeit beeinflussen. Bei der Systemanalyse stellt sich jedoch heraus, dass es schwierig sein kann, sinnvolle Größen zu finden. Warum das so ist, soll im Folgenden erläutert werden.
Generell kann man bei der Komplexitätsanalyse in einem ersten Schritt systematisch vorgehen und die Parameter der Problembeschreibung betrachten. Von diesen beeinflussen oft nur sehr wenige essentiell die Laufzeit. In diesem Fall ist das Problem durch eine Beschreibung des Systems mit allen Tasks, Verarbeitungseinheiten, Service- und Ankunftskurven gegeben. Da die in dieser Arbeit beschriebenen Berechnungen immer nur für einen Taskknoten durchgeführt werden, ist es offensichtlich, dass die Anzahl der Tasks ein grundlegender Faktor für die Laufzeitabschätzung ist.
Wenn man nur die Anzahl der Tasks als Kriterium verwendet, so muss man für die Bestimmung der asymptotischen Worst Case Laufzeit davon ausgehen, dass die Tasks durch die ungünstigste Konstellation von Ankunfts- und Servicekurven miteinander verbunden sind. Die einzigen Beschränkungen bestehen darin, dass der Graph keine Zyklen aufweisen darf und nur Scheduler mit statischen Prioritäten verwendet werden (vgl. Abschnitt 3.3). Jeder Task hat daher immer nur genau eine ausgehende Servicekurve, die nur zu genau einem weiteren Task führt, der wiederum nur diese eingehende Servicekurve besitzt. Außerdem soll es keine zwei Ankunftskurven geben, die sowohl einen identischen Quell- als auch Ziel-Task haben, da eine solche Verwendung zum Einen in der Praxis sinnlos ist und zum Anderen durch die Skalierung einer der beiden Kurven zu einer zusammengefasst werden kann (s.Abbildung 5.14).
Die Einschränkungen bezüglich der Wahl der Scheduler und der Zyklenfreiheit sind nicht unerheblich. Daher stellt sich die Frage, inwieweit die genaue Bestimmung der Laufzeitkomplexität unter diesen Einschränkungen nützlich ist. Da sie sich zudem als äußert schwierig erweist, wird hier nur darauf eingegangen, welche Laufzeitcharakteristika durch verschiedene Teilkonstrukte entstehen. Für die genaue Komplexitätsbestimmung soll hier lediglich aufgezeigt werden, worin die Probleme der Berechnung bestehen.
Ein Problem zeigt sich, wenn man die Laufzeit hauptsächlich in Abhängigkeit von der Anzahl der Tasks ermitteln möchte. Hier ist nun der schlimmstmögliche Fall gesucht. Dieser tritt ein, wenn die Laufzeiten der Tasktransformations-Berechnungen für alle Tasks in der Summe maximal sind. Wie aus dem vorherigen Abschnitt ersichtlich, müssen dafür die Testlisten der eingehenden Kurven für die Tasks maximale Länge haben. Bei nur einem Task ist dieser Fall trivial zu konstruieren, wie in Abbildung 5.15a bei τ_1 zu sehen ist. Fügt man nun

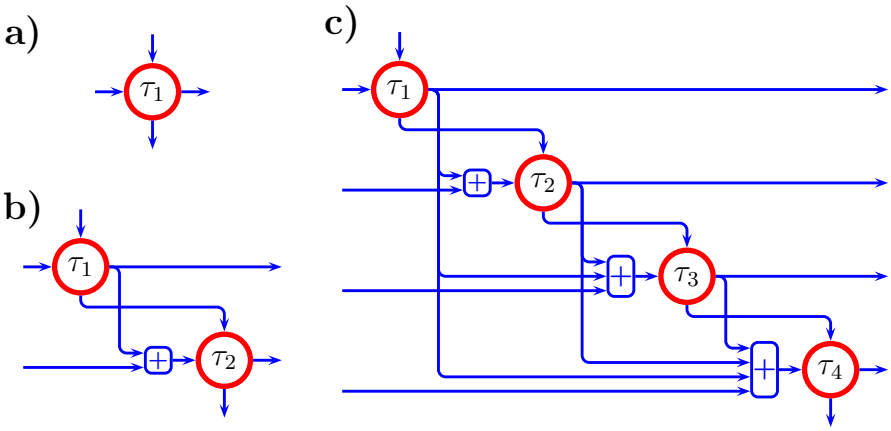


Abbildung 5.15: Worst Case Verknüpfungen bei a) einem Task, b) zwei Tasks und c) vier Tasks

noch einen weiteren Task τ_2 hinzu, so muss dieser so angeordnet werden, dass er möglichst lange Eingabelisten erhält. Das sind in diesem Fall die ausgehenden Testlisten von τ_1. Da ein Task jedoch mehrere Ankunftskurven besitzen kann, können auch externe Taskaktivierungen hinzugefügt werden (s. Abbildung 5.15b). In einem solchen System sind unter der Worst Case Annahme die ausgehenden Testlisten von τ_2 die längsten Listen. Sie können aber nicht als Eingabe des vorherigen Tasks τ_1 dienen, da sonst ein Zyklus im Graph entsteht. Sollte jedoch noch ein weiterer Task τ_3 dem System hinzugefügt werden, so kann die ausgehende Servicekurve von τ_2 als Eingabe von τ_3 dienen, da sie die längste im System ist. Für die eingehenden Ankunftskurven des neuen Tasks können dagegen alle ausgehenden Ankunftskurven der vorherigen Tasks, sowie die externen Ankunftskurven dienen. Daher ergibt sich für den schlimmsten Fall eine verkettete Struktur, wie sie in Abbildung 5.15c zu sehen ist.

Mit den bisher gemachten Annahmen spricht theoretisch nichts gegen eine solche Konstruktion. Es ist jedoch höchst fraglich, ob ein solcher Fall jemals auch nur ansatzweise in der Praxis auftreten wird. Aus der Abbildung 5.15c ist zu erahnen, dass die Anzahl der Eingänge der Additionsblöcke vor den Taskknoten mit der Anzahl der Tasks steigt. In der Praxis dürfte jedoch zu erwarten sein, dass ein Task meistens von nur einer oder sehr wenigen Quellen aus aktiviert wird. Daher bietet es sich an, die maximale Anzahl der eingehenden Ankunftskurven eines Taskknotens als Parameter für die Komplexitätsfunktion zu verwenden. In Abbildung 5.16 ist der schlimmste Fall dargestellt, wenn jeder Task nur eine Ankunftskurve hat.

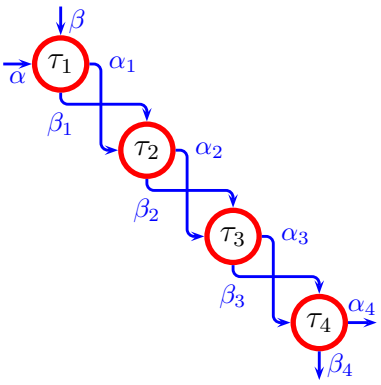


Abbildung 5.16: Worst Case Verknüpfungen mit maximal einer eingehenden Ankunftskurve

Mit den Formeln aus Tabelle 5.3 lassen sich nun für dieses Beispiel die maximalen Längen aller beteiligten Testlisten berechnen. Damit lässt sich dann auch die Worst Case Laufzeit ermitteln. Aber selbst für den relativ einfachen Fall, der in Abbildung 5.16 zu sehen ist, werden die Formeln, die sich für die einzelnen Testlisten ergeben, sehr schnell unübersichtlich.
Nachfolgend sind die Formeln für die Längen der ausgehenden Kurven der ersten zwei Tasks zu sehen. Dabei bezeichnen a_l und a_u die Längen der oberen und unteren initialen Ankunftskurve α; entsprechend sind b_l und b_u für die initiale Servicekurve β definiert. Für jede weitere Ankunftskurve α_i oder Servicekurve β_i werden diese Werte durch $a_{l,i}$ und $a_{u,i}$ bzw. $b_{l,i}$ und $b_{u,i}$ dargestellt.

$$
\begin{array}{rcl}
a_{l,1} & = & b_l a_l \cdot \log_2 (4 b_l a_l) \\
a_{u,1} & = & 2 b_l b_u a_u \cdot \log_2 (4 b_l b_u a_u) \\
b_{l,1} & = & b_l + 2 a_u \\
b_{u,1} & = & b_u + 2 a_l \\
\hline
a_{l,2} & = & (b_l + 2a_u) \cdot b_l a_l \cdot \log_2 (4 b_l a_l) \cdot \log_2 \big(4(b_l + 2a_u) \cdot b_l a_l \cdot \log_2 (4 b_l a_l)\big) \\
a_{u,2} & = & 2(b_l + 2a_u)(b_u + 2a_l) \cdot 2 b_l b_u a_u \cdot \log_2 (4 b_l b_u a_u) \\
 & & \cdot \log_2 \big(4(b_l + 2a_u)(b_u + 2a_l) \cdot 2 b_l b_u a_u \cdot \log_2 (4 b_l b_u a_u)\big) \\
b_{l,2} & = & b_l + 2a_u + 2 \cdot 2 b_l b_u a_u \cdot \log_2 (4 b_l b_u a_u) \\
b_{u,2} & = & b_u + 2a_l + 2 \cdot b_l a_l \cdot \log_2 (4 b_l a_l) \\
\hline
\vdots & & \\
\end{array}
$$

Man sieht sehr gut, dass diese Formeln sehr schnell wachsen und es nicht einfach ist, sie zusammenzufassen und eine generelle, nicht iterative Formel für $a_{l,i}, a_{u,i}, b_{l,i}$ und $b_{u,i}$ herzuleiten. Dennoch lässt sich hier einiges ablesen.
Der unübersichtliche Aufbau der Formeln rührt zu einem großen Teil von den verschachtelten Logarithmusfunktionen her. Wenn nun der logaritmische Anteil der Formeln mit 1 gleichgesetzt wird, so würden diese Formeln nicht mehr den schlimmsten Fall beschreiben. Allerdings lässt sich selbst dann noch zeigen, dass nach diesen Formeln die Testlisten ein exponentielles Wachstum aufweisen. Mit den angesprochenen, abgeänderten Formeln ergeben sich in die maximalen Längen der ausgehenden Ankunftskurven durch:

$$
\begin{array}{rcll}
a_{l,i+1} & = & b_{l,i} \cdot a_{l,i} & (5.14) \\
a_{u,i+1} & = & 2 \cdot b_{l,i} b_{u,i} \cdot a_{u,i} & (5.15)
\end{array}
$$

Berechnet man mit diesen Formeln die Längen der Listen im Beispiel so ergibt sich nun:

$$
\begin{array}{rcl}
a_{l,1} & = & b_l a_l \\
a_{u,1} & = & 2 b_l b_u a_u \\
b_{l,1} & = & b_l + 2 a_u \\
b_{u,1} & = & b_u + 2 a_l \\
\hline
\end{array}
$$

$$
\begin{array}{rcl}
a_{l,2} & = & (b_l + 2a_u) \cdot b_l a_l \\
 & = & b_l^2 a_l + 2 b_l a_l a_u \\
a_{u,2} & = & 2(b_l + 2a_u) \cdot (b_u + 2a_l)(2 b_l b_u a_u) \\
 & = & 2^2 b_l^2 b_u^2 a_u + 2^3 b_l^2 a_l a_u + 2^3 b_l b_u^2 a_u^2 + 2^4 b_l b_u a_l a_u^2 \\
b_{l,2} & = & b_l + 2a_u + 2^2 b_l b_u a_u \\
b_{u,2} & = & b_u + 2a_l + 2 b_l a_l \\
\hline
a_{l,3} & = & (b_l + 2a_u + 2^2 b_l b_u a_u)(b_l^2 a_l + 2 b_l a_l a_u) \\
 & = & b_l^3 a_l + 2 b_l^2 a_l a_u + 2^2 b_l^3 b_u a_l a_u + 2^3 b_l^2 b_u a_l a_u^2 \\
a_{u,3} & = & 2(b_l+2a_u+2^2 b_l b_u a_u)(b_u+2a_l+2 b_l a_l)(2^2 b_l^2 b_u a_u + 2^3 b_l b_u a_l a_u + 2^3 b_l b_u a_u^2 + 2^4 b_l b_u a_l a_u^2) \\
 & = & 2^3 b_l^3 b_u^2 a_u + 2^4 b_l^2 b_u^2 a_l^2 a_u + 2^8 b_l^2 b_u^2 a_u^2 + 2^5 b_l^2 b_u^2 a_l a_u^2 + 2^4 b_l^3 b_u a_l a_u + 2^5 b_l^2 b_u a_l^2 a_u + 2^{10} b_l^2 b_u a_l a_u^2 \\
 & & + 2^{12} b_l^2 b_u a_l^2 a_u^2 + 2^4 b_l^4 a_l a_u + 2^5 b_l^3 b_u a_l^2 a_u + 2^{10} b_l^3 b_u a_l a_u^2 + 2^6 b_l^3 b_u a_l^2 a_u^2 + 2^5 b_l b_u^2 a_l a_u^2 \\
 & & + 2^5 b_l b_u^2 a_u^3 + 2^6 b_l b_u^2 a_l a_u^3 + 2^6 b_l b_u a_l^2 a_u^2 + 2^6 b_l b_u a_l a_u^3 + 2^7 b_l b_u a_l^2 a_u^3 + 2^6 b_l^2 b_u a_l a_u^3 + 2^7 b_l^2 b_u a_l^2 a_u^3 \\
 & & + 2^5 b_l^3 b_u^3 a_u^2 + 2^6 b_l^2 b_u^3 a_l a_u^2 + 2^6 b_l^2 b_u^3 a_u^3 + 2^7 b_l^2 b_u^3 a_l a_u^3 + 2^6 b_l^3 b_u^2 a_l a_u^2 + 2^7 b_l^2 b_u^2 a_l^2 a_u^2 \\
 & & + 2^7 b_l^2 b_u^2 a_l a_u^3 + 2^8 b_l^2 b_u^2 a_l^2 a_u^3 + 2^6 b_l^4 b_u^2 a_l a_u^2 + 2^7 b_l^3 b_u^2 a_l^2 a_u^2 + 2^7 b_l^3 b_u^2 a_l a_u^3 + 2^8 b_l^3 b_u^2 a_l^2 a_u^3 \\
b_{l,3} & = & b_l + 2a_u + 2^2 b_l b_u a_u + 2^3 b_l^2 b_u^2 a_u + 2^4 b_l^2 a_l a_u + 2^4 b_l b_u^2 a_u^2 + 2^5 b_l b_u a_l a_u^2 \\
b_{u,3} & = & b_u + 2a_l + 2 b_l a_l + 2 b_l^2 a_l + 2^2 b_l a_l a_u \\
\hline
\vdots & & \\
\end{array}
$$

Auch diese Formeln werden schnell unübersichtlich, jedoch erkennt man gut, dass mit den hier gemachten Annahmen über den schlimmsten Fall das Wachstum der Testlisten nicht polynomiell begrenzt ist. In der Formel für $a_{u,3}$ kommt a_u zum Teil mit dem Exponenten 3 vor, was auf ein Wachstum von etwa a_u^n schließen lässt. Wenn man die Formeln jedoch noch weitere Iterationen verfolgt, so stellt sich sogar ein Wachstum von etwa $a_u^{(2^n)}$ heraus. Dabei wurden diese Ergebnisse mit vereinfachten Formeln erzielt, bei denen der logarithmische Anteil entfernt wurde. Daher werden auch die Originalformeln ein exponentielles Wachstum erzeugen.

Das Ergebnis ist nicht sehr aufmunternd. An dieser Stelle bleiben jedoch noch einige Fragen ungeklärt. Vor allem wurde nicht gezeigt, wie gut diese Abschätzung ist. Hierbei kann eine Konstruktionsvorschrift helfen, die beliebig lange Testlisten für die initialen Ankunftskurven erzeugt, so dass die Längen der daraus berechneten Kurven ein exponentielles Wachstum aufweisen.
Um so eine Vorschrift zu finden, sollte man sich erst einmal auf die Suche nach der Ursache des exponentiellen Wachstums machen. Betrachtet man die Formeln zur Berechnung der maximalen Testlistenlängen in Tabelle 5.3, so liegt die Vermutung nah, dass der Grund hauptsächlich in der Berechnung der Ankunftskurven liegt, da bei ihnen die maximale Länge aus dem Produkt der Längen der Ankunfts- und Servicekurven gebildet wird. Bei den Servicekurven wird hieraus lediglich die Summe gebildet.
Diese Vermutung wird bestärkt, wenn man sich die Listenlängen für Fälle anschaut, in denen die Tasks entweder nur über ihre Servicekurven oder nur über ihre Ankunftskurven miteinander verknüpft sind.

5.5.2 Einfluss der Servicekurven

Sind die Tasks nur durch ihre Servicekurven voneinander abhängig, ergibt sich mit den hier getroffenen Annahmen das Beispiel eines Einprozessorsystems. Dies ist in Abbildung 5.17 zu sehen.
Es sollen nun die maximalen Längen der beteiligten Listen ermittelt werden. Dabei bezeichnen auch hier $a_{l,i}, a_{u,i}, b_{l,i}$ und $b_{u,i}$ die maximalen Längen der Testlisten von α_i und β_i. Die initiale Servicekurve des Prozessors ist β_0. Sie wird für beide Grenzen durch ein einzelnes Testlisten-Element dargestellt, das einen Gradienten mit Steigung 1 beschreibt. Mit den bekannten Formeln ergibt sich daher für die weiteren Servicekurven:

$$
\begin{array}{rcl}
b_{l,1} & = & 1 + 2a_{u,1} \\
b_{u,1} & = & 1 + 2a_{l,1} \\
\hline
b_{l,2} & = & 1 + 2a_{u,1} + 2a_{u,2} \\
b_{u,2} & = & 1 + 2a_{l,1} + 2a_{l,2} \\
\hline
b_{l,3} & = & 1 + 2a_{u,1} + 2a_{u,2} + 2a_{u,3} \\
b_{u,3} & = & 1 + 2a_{l,1} + 2a_{l,2} + 2a_{l,3} \\
\hline
 & \vdots & \\
\hline
b_{l,n} & = & 1 + 2\sum_{i=1}^{n} a_{u,i} \\
b_{u,n} & = & 1 + 2\sum_{i=1}^{n} a_{l,i} \\
\hline
\end{array}
$$

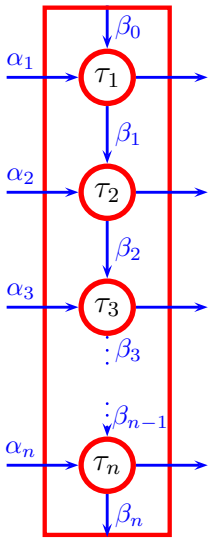


Abbildung 5.17: Einprozessorsystem

Anstatt die Werte für jede Ankunftskurve zu berücksichtigen, kann eine Überapproximation vorgenommen werden, indem als Länge für jede Ankunftskurve die maximale Länge aller Ankunftskurve verwendet wird:

$$a_l = \max\{a_{l,i},\ \ 1 \leq i \leq n\} \qquad\qquad a_u = \max\{a_{u,i},\ \ 1 \leq i \leq n\}$$

Eine solche Vorgehensweise ist insbesondere bei periodischen Tasks sehr praxisnah. Werden für alle Tasks die gleiche Anzahl an Perioden exakt dargestellt, so haben tatsächlich alle Ankunftskurven die gleiche Länge. In diesem Fall würden a_l und a_u den Grad der Approximation angeben. Mit dieser Annahme ergibt sich für die maximalen Längen der Servicekurven:

$$b_{l,i} = 1 + 2 \cdot i \cdot a_u \qquad\qquad b_{u,i} = 1 + 2 \cdot i \cdot a_l$$

Damit und mit der Komplexitätsfunktion für die Laufzeit einer Tasktransformation aus Gleichung 5.12 kann nun eine Abschätzung für die Gesamtlaufzeit aller Tasktransformationen gemacht werden. Dazu werden die Laufzeiten aller

n Tasktransformationen addiert:

$$
\begin{aligned}
& \sum_{i=0}^{n-1} \left(\left((\log_2 a_l)^2 + (\log_2 b_{l,i})^2 \right) \cdot b_{l,i} a_l + \left((\log_2 b_{l,i})^2 + \left(\log_2 (a_u b_{u,i}) \right)^2 \right) \cdot \log_2 (a_u b_{u,i}) \cdot a_u b_{u,i} b_{l,i} \right) \\
= & \sum_{i=0}^{n-1} \left(\left((\log_2 a_l)^2 + \left(\log_2 (1+2ia_u) \right)^2 \right) \cdot (1+2ia_u) a_l \right. \\
& \qquad \left. + \left(\left(\log_2 (1+2ia_u) \right)^2 + \left(\log_2 (a_u(1+2ia_l)) \right)^2 \right) \cdot \log_2 \left(a_u(1+2ia_l) \right) \cdot a_u(1+2ia_l)(1+2ia_u) \right) \\
< & \sum_{i=0}^{n-1} \left(\left((\log_2 a_l)^2 + \left(\log_2 (3ia_u) \right)^2 \right) \cdot (3ia_u) a_l \right. \\
& \qquad \left. + \left(\left(\log_2 (3ia_u) \right)^2 + \left(\log_2 (a_u(3ia_l)) \right)^2 \right) \cdot \log_2 \left(a_u(3ia_l) \right) \cdot a_u(3ia_l)(3ia_u) \right) \\
< & \sum_{i=0}^{n-1} \left(\left(\log_2 (3ia_l a_u) \right)^2 \cdot 3ia_u a_l + \left(\log_2 (3^2 i^2 a_l a_u) \right)^2 \cdot \log_2 (3ia_l a_u) \cdot 3^2 i^2 a_l a_u^2 \right) \\
< & \sum_{i=0}^{n-1} \left(\left(\log_2 (3ia_l a_u) \right)^2 \cdot 3ia_u a_l + \left(\log_2 (3^2 i^2 a_l a_u) \right)^3 \cdot 3^2 i^2 a_l a_u^2 \right) \\
< & \sum_{i=0}^{n-1} \left(\left(\log_2 (3ia_l a_u) \right)^2 \cdot 3ia_u a_l + 4 \left(\log_2 (3ia_l a_u) \right)^3 \cdot 3^2 i^2 a_l a_u^2 \right) \\
< & \sum_{i=0}^{n-1} \left(1000 ia_l a_u \cdot 3ia_u a_l + 1000 ia_l a_u \cdot 3^2 i^2 a_l a_u^2 \right) \\
= & \sum_{i=0}^{n-1} (i^2) \cdot 3000 a_l^2 a_u^2 + \sum_{i=0}^{n-1} (i^3) \cdot 9000 a_l^2 a_u^3 \\
= & \frac{2n^3 - 3n^2 + n^2}{6} \cdot 3000 a_l^2 a_u^2 + \frac{n^4 - 2n^3 + n^2}{4} \cdot 9000 a_l^2 a_u^3
\end{aligned}
$$

Für die Angabe der Komplexitätsklasse ist wiederum nur der am stärksten wachsende Summand ohne konstante Faktoren interessant. Daher liegt die Komplexität bei der hier betrachteten Taskverknüpfung in $O(n^4 a_u^3 a_l^2)$ und ist somit polynomiell begrenzt. Da es sich dabei um ein Einprozessorsystem handelt, das mit statischen Prioritäten geplant wird, kann so auch gezeigt werden, dass die approximierte Analyse in einem solchen Fall in polynomieller Zeit durchgeführt werden kann.

5.5.3 Einfluss der Ankunftskurven

Mit dem Ergebnis aus dem letzten Abschnitt ist nun offensichtlich, dass die Berechnung der Servicekurven nicht die alleinige Ursache für das exponentielle Wachstum der Testlisten aus dem ersten Beispiel sein kann. Ähnlich wie beim Einprozessorsystem soll nun betrachtete werden, was sich ergibt, wenn die Tasks nur über ihre Ankunftskurven miteinander verknüpft werden.
Wenn man nun versucht, ein System analog zu dem Beispiel im vorherigen Abschnitt aufzubauen, so ergibt sich ein System, wie es in Abbildung 5.18 zu sehen

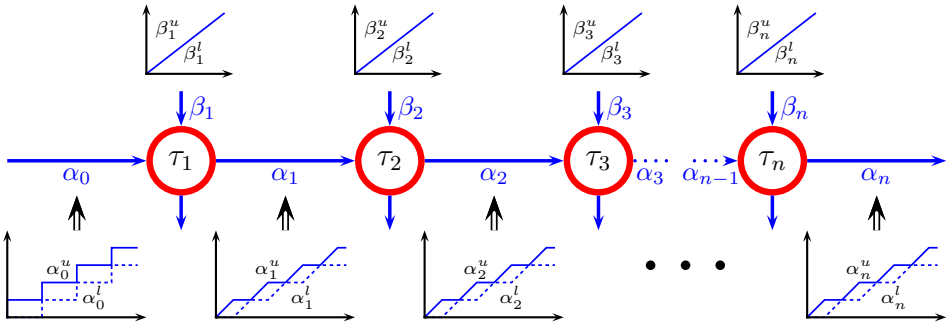


Abbildung 5.18: Verknüpfung der Tasks ausschließlich über ihre Ankunftskurven

ist. Da Beziehungen nur über Ankunftskurven hergestellt werden, erhält jeder Task die initiale Servicekurve mit Steigung 1 als Eingabe. Daher ergibt sich im Hinblick auf das Beispiel des Einprozessorsystems hier genau der entgegengesetzte Extremfall, in dem jeder Task auf einem separaten Prozessor untergebracht ist. Es zeigt sich allerdings, dass ein solches System ungünstig ist, um das exponentielle Wachstum der Formeln zu untersuchen. Die initialen Servicekurven stellen einen sehr speziellen Fall dar. Sie bewirken, dass sich die Ankunftskurven nur bei der ersten Tasktransformation ändern. Danach sind für jeden Taskknoten eingehende und ausgehende Ankunftskurven identisch. Demnach lässt sich so auch kein exponentielles Wachstum dieser Kurven finden.
Dies lässt sich durch eine leichte Anpassung des Beispiels ändern. Dabei werden nun jeweils zwei der n Tasks auf jeden Prozessor untergebracht. Die Verknüpfung durch die Ankunfts- und Servicekurven erfolgt wie in Abbildung 5.19 dargestellt.

Die Bezeichnung der Testlistenlängen erfolgt wie bei den vorherigen Beispielen. Auch hier soll für die Berechnungen wieder davon ausgegangen werden, dass alle Ankunftskurven, die als Eingabe für das System dienen, die gleiche Längen a_l bzw. a_u haben.
Bei der Untersuchung des Einflusses der Servicekurven wurde eine Obergrenze für die Laufzeit bei der Analyse von Einprozessorsystemen ermittelt. Das ist auch sinnvoll, da solche Systeme für die Praxis relevant sind. Die System-

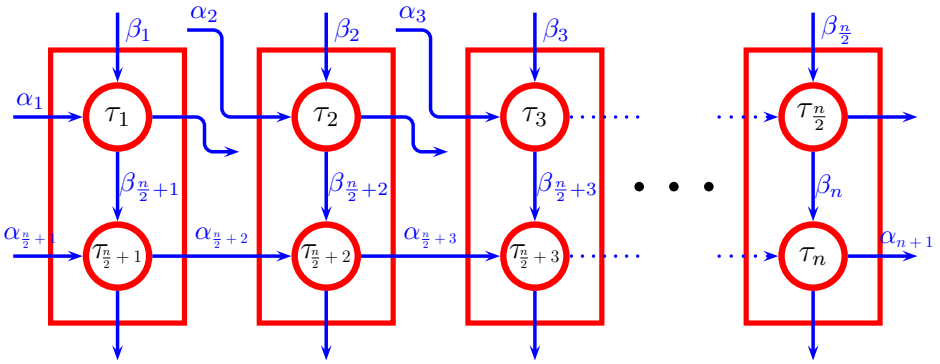


Abbildung 5.19: Durch die Konstruktionsvorschrift gegebener Taskgraph

Konstruktion in Abbildung 5.19 ist jedoch praktisch wahrscheinlich weniger relevant. Daher ist hier auch keine Obergrenze für die Laufzeit gesucht. Es kann nun jedoch mit diesem generischen Tasksystem leicht eine Vorschrift angegeben werden, die für eine beliebige Länge der initialen Ankunftskurven α_1 bis $\alpha_{\frac{n}{2}+1}$ und eine beliebige Anzahl an Tasks die Ankunftskurven so generiert, dass die Laufzeit der Systemanalyse exponentiell von der Anzahl der Tasks abhängt. Diese Konstruktionsvorschrift soll nun dargestellt werden.
Die initialen Servicekurven β_1 bis $\beta_{\frac{n}{2}}$ bestehen aus einem einzigen Segment der Steigung 1 und haben daher die Länge 1. Die Ankunftskurven α_1 bis $\alpha_{\frac{n}{2}+1}$ stellen jeweils periodische Taskaktivierungen dar, die für a Perioden exakt dargestellt werden und anschließend approximiert werden. Daher haben die oberen Grenzen dieser Kurven eine Länge von $a+1$. Der generische Aufbau dieser Ankunftskurven ist in Abbildung 5.20a für $a = 3$ dargestellt.
Der Task $\tau_{\frac{n}{2}}$ hat eine besondere Rolle seine Periode und Ausführungszeit ist frei wählbar. Seine Periode wird im weitere mit $T_{\frac{n}{2}}$ und die Ausführungszeit mit $C_{\frac{n}{2}}$ bezeichnet. Für die Perioden und die Ausführungszeiten der Tasks τ_1 bis $\tau_{\frac{n}{2}-1}$ soll nun gelten:

$$T_i = a \cdot T_{i+1} + \frac{C_{i+1}}{1 - \frac{C_{i+1}}{T_{i+1}}} \tag{5.16}$$

$$C_i = a \cdot C_{i+1} + \frac{C_{i+1}}{1 - \frac{C_{i+1}}{T_{i+1}}} \tag{5.17}$$

Desweiteren soll für $\tau_{\frac{n}{2}+1}$ gelten:

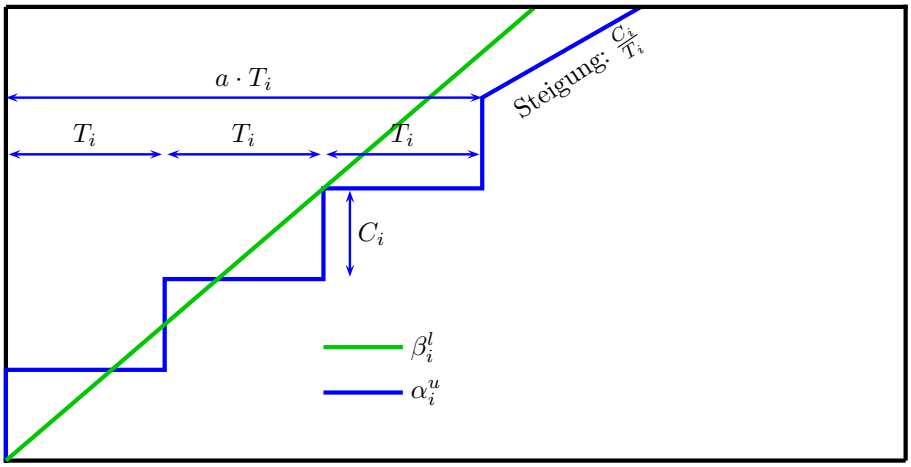
$$T_{\frac{n}{2}+1} = a \cdot T_1 + \frac{C_1}{1 - \frac{C_1}{T_1}} \tag{5.18}$$

$$C_{\frac{n}{2}+1} = a \cdot C_1 + \frac{C_1}{1 - \frac{C_1}{T_1}} \tag{5.19}$$

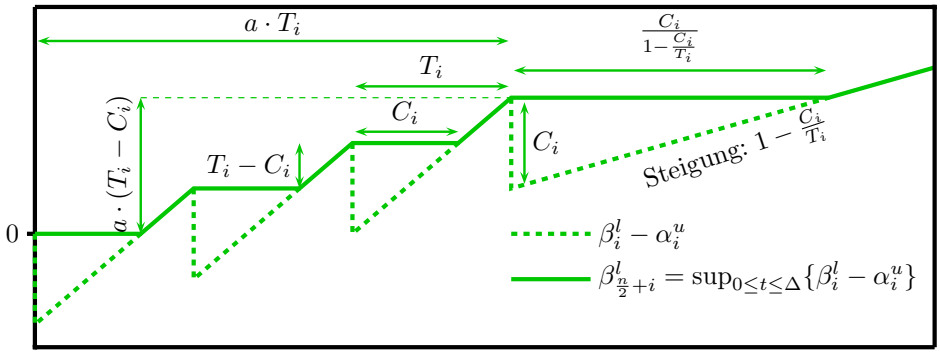
Berechnet man nun für die Tasks von τ_1 bis $\tau_{\frac{n}{2}}$ die unteren ausgehenden Ankunftskurven, so zeigt sich, dass diese immer eine Länge von $2a+1$ haben (vgl. Abschnitt 5.4.2):

$$\beta_{l,i} = 2a + 1, \qquad \frac{n}{2} < i \leq n \tag{5.20}$$

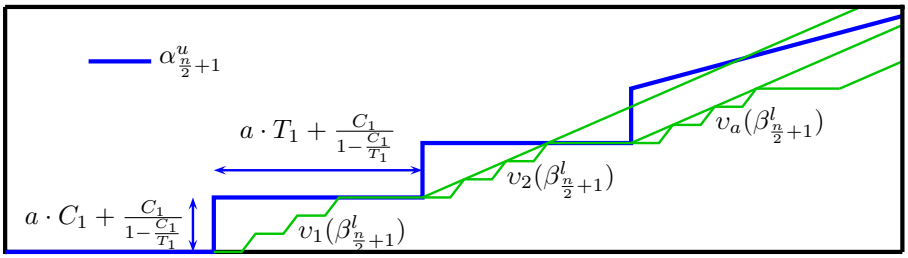
Der Aufbau dieser Kurven ist ebenfalls für $a = 3$ in Abbildung 5.20b dargestellt. Dort sind auch alle Längen eingetragen, die für die weiteren Überlegungen notwendig sind.
In der Abbildung 5.20c ist die Berechnung der unteren Ankunftskurve von $\tau_{\frac{n}{2}+1}$ grafisch dargestellt. Dazu sind die verschobenen Verläufe $\upsilon_1(\beta^l_{\frac{n}{2}+i})$ bis $\upsilon_a(\beta^l_{\frac{n}{2}+i})$ der eingehenden unteren Servicekurve eingezeichnet. Da die verschobenen Verläufe der Ankunftskurven in diesem Fall keine Auswirkungen auf das Ergebnis haben, sind sie nicht in der Zeichnung abgebildet. Hier lässt sich nun die Begründung für die Gleichungen 5.18 und 5.19 erkennen. Die Ankunftskurve $\alpha^l_{\frac{n}{2}+1}$ ist eine Treppenfunktion. Durch die obige Wahl der Periode und Ausführungszeit wird jede Treppenstufe so dimensioniert, dass der Verlauf der eingehenden unteren Servicekurve vom ersten bis zum letzten Testlisten-Element genau einmal unter eine Treppenstufe passt. Da die Servicekurve an den Beginn jeder Treppenstufe geschoben wird, ergibt sich für die ausgehende unterer Ankunftskurve



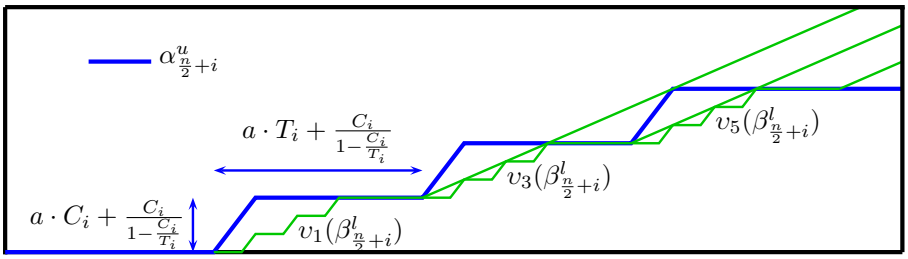
a) Aufbau der eingehenden oberen Ankunfts- und unteren Servicekurven der Tasks τ_1 bis $\tau_{\frac{n}{2}}$



b) Aufbau der ausgehenden unteren Servicekurven der Tasks τ_1 bis $\tau_{\frac{n}{2}}$



c) Berechnung der ausgehenden unteren Ankunftskurven von $\tau_{\frac{n}{2}+1}$



d) Berechnung der ausgehenden unteren Ankunftskurven von $\tau_{\frac{n}{2}+i}$ für $i > 1$

Abbildung 5.20: Aufbau der Kurven in der Konstruktionsvorschrift

eine Länge von $a_{l,\frac{n}{2}+1} = a \cdot 2a$.
Bei Betrachtung der Abbildung 5.20b erkennt man, dass auch die eingehenden Servicekurven für die Tasks $\tau_{\frac{n}{2}+1}$ bis τ_n eine Art Treppenfunktion mit abgeschrägten Stufen beschreiben. Die Servicekurven haben dabei bis zum letzten Testlisten-Element immer a Stufen. Die a Stufen der Servicekurve $\alpha^l_{\frac{n}{2}+1}$ finden sich daher auch in Abbildung 5.20c unter jeder der a Stufen der Ankunftskurve $\alpha^l_{\frac{n}{2}+1}$. Somit hat die berechnete, ausgehende untere Ankunftskurve $\alpha^l_{\frac{n}{2}+2}$ also $a \cdot a$ Stufen und ihre Testliste daher $2a^2$ Elemente. Aufgrund der Gleichungen 5.16 und 5.17 sind diese Stufen wiederum so dimensioniert, dass der Verlauf der unteren Servicekurve $\beta^l_{\frac{n}{2}+2}$ bis zum letzten Testlisten-Element genau einmal unter einer dieser Stufen passt. Dies ist in Abbildung 5.20d dargestellt. Die Berechnung der unteren ausgehenden Ankunftskurve von $\tau_{\frac{n}{2}+2}$ verhält sich also wie die bei $\tau_{\frac{n}{2}+1}$. Dies gilt auch für alle weiteren Tasks.
Daher ergeben sich für die Ankunftskurven $\alpha^l_{\frac{n}{2}+1}$ bis α^l_n folgende Längen:

$$\begin{aligned}
a_{l,\frac{n}{2}+1} &= a \\
a_{l,\frac{n}{2}+2} &= a \cdot a \\
&\vdots \quad\quad \vdots \\
a_{l,i} &= a^{i-\frac{n}{2}} \\
&\vdots \quad\quad \vdots \\
a_{l,n-1} &= a^{\frac{n}{2}-1} \\
a_{l,n} &= a^{\frac{n}{2}}
\end{aligned} \tag{5.21}$$

Anhand der Gleichung 5.21 ist nun ersichtlich, dass bei der hier gegebenen Konstruktionsvorschrift die Länge der eingehenden unteren Ankunftskurve exponentiell von der Anzahl der Tasks abhängig ist. Da die Algorithmen zur Berechnung der ausgehenden Kurven immer über die vollständigen Testlisten iterieren, ist damit auch gezeigt, dass bereits die Laufzeit zum Iterieren der Listen exponentiell von der Anzahl der Tasks abhängt.
In den meisten vorherigen Abschnitten wurde zur Laufzeitabschätzung eine Obergrenze mittels der O-Notation angegeben. Mit dem hier gewählten Vorgehen ist als Ergebnis dagegen gezeigt, dass ein exponentielles Wachstum in der Praxis erwartet werden muss, da hier Systeme gefunden wurden, die dieses Wachstum aufweisen.
Bei einem solchen Vorgehen wird durch die Konstruktionsvorschrift jedoch nur eine sehr kleine und spezielle Teilmenge der möglichen Systeme betrachtet. Bei Aussagen über die in der Praxis zu erwartenden Laufzeiten sollte daher diese Teilmenge bezüglich ihrer praktischen Relevanz bewertet werden.
Der prinzipielle Aufbau der Systeme, so wie es in Abbildung 5.19 dargestellt ist, ist hauptsächlich auf diese Weise angelegt worden, um eine verkettete Aktivierung von Tasks auf verschiedenen Verarbeitungseinheiten zu erreichen. Das ist so oder so ähnlich bei verschiedenen Anwendungen durchaus häufiger zu erwarten. Zudem treten in der Praxis zu einem großen Teil auch periodische Tasks auf. Die Relevanz der weiteren Taskparameter ist dagegen fraglicher. Durch die Gleichungen 5.16 und 5.17 ergibt sich eine exponentielle Verteilung der Perioden und Ausführungszeiten der Tasks. Zudem ist diese Verteilung entlang des Kontroll- und Datenflusses des Systems geordnet. Das ist eine sehr spezielle Ei-

genschaft, die bei durchschnittlichen Systemen nicht zu erwarten ist.
An dieser Stelle bieten sich daher verschiedene weiterführende Maßnahmen an, die in zukünftigen Arbeiten behandelt werden können. So können die Bedingungen, die ein exponentielles Wachstum erzeugen, detailierter untersucht werden, um entweder eine Konstruktionsvorschrift zu finden, die eine höhere praktische Relevanz hat und ebenfalls ein exponentielles Wachstum ergibt, oder zu zeigen, dass das exponentielle Wachstum eben nur in Sonderfällen auftritt. Zur weiteren Untersuchung ist auch eine statistische Analyse sinnvoll, um durchschnittliche Werte zu erhalten und Aussagen über verschiedene Wahrscheinlichkeiten treffen zu können.

Es zeigte sich bei der Komplexitätsuntersuchung einer Systemanalyse, dass generell davon auszugehen ist, dass die Laufzeit exponentiell anwachsen kann. Bei den genaueren Betrachtungen zum Einfluss der Service- und Ankunftskurven stellt sich allerdings heraus, dass die Servicekurven nicht der ausschlaggebende Faktor für das exponentielle Wachstum sind. Es wurde sogar gezeigt, dass die Analyse eines Einprozessorsystems in polynomieller Zeit durchgeführt werden kann, wenn die Tasks nur über ihre Servicekurven miteinander verbunden sind. Dagegen ergibt sich durch die gegebenen Konstruktionsvorschrift, dass beliebig große System erzeugt werden können, die aufgrund ihrer Ankunftskurven eine exponentiell wachsende Laufzeit aufweisen.
Um eine genauerer Aussage über die Laufzeit einer Systemanalyse zu erhalten, bietet es sich daher für zukünftige Arbeiten an, eine Obergrenze für die Laufzeit zu finden, die als einen Parameter die Tiefe des Taskgraphen in Richtung der Ankunftskurven hat.

Kapitel 6

Fazit

In dieser Arbeit wurden die in [] entwickelten Algorithmen in einem Java-Framework umgesetzt. Anhand dieser Implementierung wurde systematische die Komplexität der einzelnen Teilfunktionen bestimmt. Dies geschah in einem ersten Schritt immer strukturiert anhand des Quellcodes. In einem weiteren Schritt wurde wenn nötig das dynamische Verhalten der zugrunde liegenden Datenstrukturen ausführlich betrachtet, um hieraus Rückschlüsse auf die maximale Laufzeit der Funktion zu ziehen. Dabei wurde die Bestimmung in Abhängigkeit von zwei verschiedenen Datenstrukturen vorgenommen, die als Grundlage der Problembeschreibung dienen. Auf diese Weise konnte die optimalere Struktur ermittelt werden.
An einigen Stellen wurden Probleme und Lösungsmöglichkeiten im Zusammenhang mit der Fließkommaarithmetik innerhalb der Algorithmen aufgezeigt. Da es in diesem Themengebiet umfangreiche Möglichkeiten zur Optimierung der Algorithmen gibt, bietet es sich an, die Implementierung in einer weiteren Arbeit mit diesem Schwerpunkt zu untersuchen und zu optimieren.
Desweiteren wurde durch die Komplexitätsanalyse ein sehr tiefer Einblick in die Arbeitsweisen und das Verhalten der unterschiedlichen Funktionen gewonnen. Als eine Kernaussage zeigte sich, dass die Berechnung einer Tasktransformation - das zentrale Konzept des Realtime Calculus - in polynomieller Zeit durchgeführt werden kann.
In einem weiteren Schritt wurde auch das Zusammenspiel mehrere Tasktransformationen ausführlich untersucht. Dabei zeigte sich, dass für einen Taskgraphen im generellen Fall die Laufzeit der Analyse exponentiell von der Größe des Graphen abhängig ist. Anhand von Beispielen und generellen Konstruktionsvorschriften konnte zudem gezeigt werden, dass die Ankunfts- und Servicekurven hierbei sehr unterschiedlichen Einfluss auf die Laufzeit haben. In diesem Zusammenhang wurde eine Klasse von realisierbaren Systemen ermittelt, bei denen durch die Verknüpfungen der Ankunftskurven die Laufzeit der Analyse exponentiell von der Anzahl der Tasks abhängt.
Die Systemanalyse bietet jedoch für zukünftige Arbeiten noch genügend Raum, um die Komplexität für verschiedene Klassen von Anwendungen und Faktoren detailiert zu untersuchen.

Literaturverzeichnis

[1] Karsten Albers, Frank Bodmann, and Frank Slomka. Hierarchical event streams and dependency graphs: A new computational model for embedded real-time systems. In *ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS'06)*, pages 97–106, Washington, DC, USA, 2006. IEEE Computer Society.

[2] Karsten Albers, Frank Bodmann, and Frank Slomka. An approximative event model for the network calculus with polynomial complexity. 2007. Internes Arbeitspapier.

[3] Karsten Albers, Frank Bodmann, and Frank Slomka. An eventstream calculus for the schedulability analysis of distributed embedded systems. Internes Arbeitspapier, 2007.

[4] Karsten Albers and Frank Slomka. Efficient feasibility analysis for real-time systems with edf scheduling. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 492–497, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Alexander Asteroth and Christel Baier. *Theoretische Informatik : eine Einführung in Berechenbarkeit, Komplexität und formale Sprachen mit 101 Beispielen*. Pearson Studium, München, 2003.

[6] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus - A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[7] Daniel Pierre Bovet and Pierluigi Crescenzi. *Introduction to the theory of complexity*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.

[8] A. Burns. Scheduling hard real-time systems: A review. *Software Engineering Journal*, pages 116+, Mai 1991. Institution of Electrical Engineers (IEE).

[9] Samarjit Chakraborty, Simon Künzli, and Lothar Thiele. Approximate schedulability analysis. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, page 159, Washington, DC, USA, 2002 IEEE Computer Society.

[10] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusets, 1st edition, 1990.

[11] Albert Einstein. Spacetime. *Encyclopædia Britannica*, 13, 1926. online: http://www.britannica.com/eb/article-9117889 (12. Okt. 2007).

[12] Klaus Gresser. *Echtzeitnachweis ereignisgesteuerter Realzeitsysteme*. PhD thesis, Technische Universität München, 1993.

[13] Daniel Jelkmann, Karsten Albers, and Frank Slomka. Improved feasibility tests for asynchronous real-time periodic task sets. In *10. GI/ITG/GMM Workshop 'Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen'*, März 2007.

[14] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[15] Dirk Nowotka. Modellierung und Analyse von Echtzeitsystemen. `http://www.fmi.uni-stuttgart.de/szs/teaching/ws0506/maes/`, letzter Zugriff: 17.11.07, 2005. Skript zur gleichnamigen Vorlesung im WS 05/06 an der Universität Stuttgart.

[16] Olaf Schenk. Algorithmen des Wissenschaftlichen Rechnens - Gleitkommaarithmetik und Fehleranalyse. `http://informatik.unibas.ch/lehre/ss03/algorechnen/online/folien/lekt3.pdf`, letzter Zugriff: 17.11.07, 2003. Skript zur Vorlesung *Algorithmen des Wissenschaftlichen Rechnens* im SS03 an der Universität Basel.

[17] Q. F Stout and B. L Warren. Tree rebalancing in optimal time and space. *Communications of the ACM*, 29(9):902–908, 1986.

[18] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systeme. In *Proceedings of the IEEE International Conference on Circuits and Systems*, 2000.

[19] Lothar Thiele, Samarjit Chakraborty, Matthias Gries, and Simon Künzli. Design space exploration of network processor architectures. In Patrick Crowley, Mark A. Franklin, Haldun Hadimioglu, and Peter Z. Onufryk, editors, *Network Processor Design: Issues and Practices*, The Morgan Kaufmann Series in Computer Architecture and Design, chapter 4, pages 55–90. Morgan Kaufmann, 2002.

[20] Lothar Thiele, Samarjit Chakraborty, Matthias Gries, Alexander Maxiaguine, and Jonas Greutert. Embedded software in network processors - models and algorithms. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 416–434, London, UK, 2001. Springer-Verlag.

[21] Ernesto Wandeler, Lothar Thiele, Marcel Verhoef, and Paul Lieverse. System architecture evaluation using modular performance analysis: a case study. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):649–667, 2006.

[22] Ingo Wegener. *Komplexitätstheorie - Grenzen der Effizienz von Algorithmen*. Springer-Verlag, Berlin, Heidelberg, 2003.

[23] Bertram Wortelen. Statistische Analyse von Scheduling- und Echtzeitanalysestrategien. Individuelles Projekt im Department für Informatik der Universität Oldenburg, Februar 2007.