Perspektiven der parallelen, ereignisgesteuerten Simulation am Beispiel von Warteschlangennetzen

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Technischen Fakultät
der Universität des Saarlandes

Dipl.-Math. Dipl.-Inform. Jörg Richter

Saarbrücken
1995

Tag des Kolloquiums:	22. Februar 1995
Dekan:	Prof. Dr. H. Bley
Berichterstatter:	Prof. Dr. F. Mattern
	Prof. Dr. R. Wilhelm

Zusammenfassung

Simulation ist ein wirkungsvolles Hilfsmittel zur Analyse komplexer, nicht vollständig mathematisch analysierbarer Zusammenhänge, das mittlerweile in weiten Bereichen von Wissenschaft und Technik Anwendung gefunden hat. Leider erwiesen sich dabei die in der Praxis auftretenden Simulationen als äußerst zeitaufwendig. Daher liegt es nahe zu untersuchen, in wieweit sich Simulationen durch "Supercomputer" beschleunigen lassen.

Diese Arbeit beschäftigt sich speziell mit der Parallelisierung ereignisgesteuerter Simulationen. Die ersten Ansätze zur Parallelisierung dieser Klasse von Simulationen liegen bereits mehr als zehn Jahre zurück. In dieser Zeit entstanden zwar zahlreiche konzeptionelle Arbeiten und kleinere Prototypen, jedoch wurde der Frage "Lohnt sich die Parallelisierung ereignisgesteuerter Simulationen?" bisher kaum systematisch nachgegangen. Die vorliegende Arbeit untersucht dies am Beispiel der Simulation von Warteschlangennetzen.

Ausgehend von einem mathematisch definierten Begriff des Simulationsmodells werden dabei zunächst die besonderen Anforderungen an ein "parallelisierbares" Modell exakt definiert, die bisher entwickelten Parallelisierungsansätze systematisch dargestellt sowie die Modellwelt und die Komponenten des eigens für diese Arbeit entwickelten Testbetts DISQUE vorgestellt. Mit diesem Testbett werden anschließend mit Hilfe von umfangreichen Messungen die Probleme bei der Realisierung eines effizienten parallelen Simulators aufgezeigt und bezüglich ihrer Bedeutung gewichtet. Hierzu wird eine Folge von teils real durchgeführten, teils aus Meßdaten berechneten, hypothetischen Simulationsläufen entwickelt, die immer realistischer werdende Sichten auf einen parallelen Simulationslauf darstellen. Die Differenz der Laufzeiten zweier Simulationsläufe der Folge ergibt dann ein Maß für die Bedeutung des zusätzlich berücksichtigten Aspekts. Weiterhin werden verschiedene parallele Synchronisationsstrategien bei der parallelen, ereignisgesteuerten Simulation bezüglich ihrer Effizienz miteinander verglichen. Den Abschluß dieser Arbeit bildet eine Zusammenfassung aller Einzelresultate im Hinblick auf den Beitrag, den die Ergebnisse dieser Arbeit zu der eingangs gestellten Frage liefern. Es zeigt sich dabei, daß zwar in Einzelfällen eine effektive Parallelisierung ereignisgesteuerter Simulationen durchaus möglich ist, sich jedoch bedingt durch die feine Granularität der meisten Anwendungen parallele, ereignisgesteuerte Simulation wohl nicht auf breiter Front durchsetzen wird.

Mein Dank gilt Herrn Prof. Dr. F. Mattern für die Betreuung meiner Promotion und für seine konstruktive Kritik an ihr, insbesondere was die vorliegende Ausarbeitung betrifft, sowie Herrn Prof. Dr. R. Wilhelm für die Übernahme des Korreferats.

Weiterhin danke ich allen "meinen" Studenten, die mit ihren Praktikums- und Diplom-Arbeiten wesentlich zum Gelingen des praktischen Teils meiner Promotion beigetragen haben.

Schließlich möchte ich noch allen danken, die mir mit Rat und Tat bei meiner Arbeit und dem täglichen Kampf mit den Computern geholfen haben.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In den letzten 30 Jahren war die praktische Informatik unter anderem durch eine stürmische Entwicklung immer leistungsfähigerer Rechner gekennzeichnet. Trotzdem existieren nach wie vor Anwendungsbereiche (beispielsweise in der Physik, der Chemie, der Biologie oder der Ökologie), deren Bedarf an Rechenleistung noch nicht einmal annähernd gedeckt ist. Darüber hinaus gibt es interessante und für die Praxis bedeutsame Probleme (wie etwa die Simulation der chemischen Vorgänge in der Atmosphäre, vgl. [Mye92]), deren Lösung durch die leistungsfähigsten, im Augenblick verfügbaren Rechner sich gerade erst abzuzeichnen beginnt.

Demgegenüber ist ein Ende der rasanten Leistungssteigerungen bei der Entwicklung neuer Prozessoren langsam absehbar. Ein Grund dafür liegt beispielsweise darin, daß bedingt durch die Eigenschaften von Metallverbindungen zwischen Rechnerplatinen und auf Multichip-Modulen die Taktraten von Prozessoren nicht beliebig steigerbar sind. Die Grenze wird bei einem Gigahertz vermutet und ihr Erreichen für die Jahrtausendwende prognostiziert. Eine Alternative wären optische Verbindungen, doch ist diese Technologie wohl in absehbarer Zeit nicht kommerziell einsetzbar und löst auch nicht alle im Zusammenhang mit Metallverbindungen auftretenden Probleme [SC91]. Ähnliches gilt für die Integrationsdichte von Elementen auf VLSI-Chips, wobei auch hier das Erreichen der technologischen Grenzen für die Jahrtausendwende prognostiziert wird [PM86]. Auch der Einsatz innovativer Architekturkonzepte beim Entwurf neuer Prozessoren dürfte früher oder später ausgereizt sein.

Die Idee, höhere Rechnerleistungen durch Parallelverarbeitung, also durch gleichzeitige Bearbeitung von Teilproblemen eines Problems mit mehreren Prozessoren zu erzielen, ist keineswegs neu[1]. Da aber Parallelrechner erst seit wenigen Jahren

[1] Die Grundidee des "parallelen Rechnens" findet sich schon bei Charles Babbage (1792-1871):

allgemein verfügbar sind, beginnen erst allmählich experimentelle Untersuchungen darüber, bei welchen Anwendungsbereichen sich tatsächlich Leistungssteigerungen mit Parallelrechnern gegenüber herkömmlichen (sequentiellen) Maschinen erzielen lassen.

Ein Großteil der informatikbezogenen Forschung auf dem Gebiet der Parallelrechner - oder allgemeiner der Mehrrechnersysteme - konzentrierte sich bisher auf Hardware- und Architekturprinzipien, auf theoretische Aspekte (parallele und verteilte Algorithmen, Berechnungsmodelle, Komplexitätsuntersuchungen etc.), auf verteilte Betriebssysteme sowie auf Werkzeuge zur Programmentwicklung (parallele und verteilte Programmiersprachen, Performance-Monitore, Debugger, etc.). Die Frage nach der praktischen Anwendbarkeit dieser Konzepte und Hilfsmittel läßt sich aber auf Grund vielfältiger synergetischer Effekte und letztlich nicht vernachlässigbarer Nebenbedingungen zumeist erst anhand von (zumindest prototypisch) implementierten Anwendungen aus der Praxis beantworten.

Auf die meisten bisher realisierten Anwendungen für Parallelrechner trifft der Begriff "Datenparallelismus" zu, wobei jeder Prozessor weitgehend unabhängig auf einem Teilraum des gesamten zu bearbeitenden Raums arbeitet. Daten werden dabei entweder mit einem zentralen Prozessor ausgetauscht, der die gesamte Anwendung steuert, oder aber mit einigen wenigen "benachbarten" Prozessoren und dies auch nur (relativ zur Gesamtlaufzeit) selten. Solche Anwendungen sind in der Regel leicht effektiv zu implementieren, insbesondere deswegen, weil hierbei viele problematische Aspekte der Programmierung auf Parallelrechnern wie beispielsweise verteilte Kontrolle zwischen autonom operierenden Prozessoren, nicht-deterministische Berechnungsabläufe, feinkörnige Granularität, fehlende globale Sicht usw. nicht oder nur in unerheblichem Maße auftreten.

Natürlich haben längst nicht alle in der Praxis auftretenden Probleme eine derartig homogene Struktur. Zu den wenigen zumindest ansatzweise untersuchten Anwendungen, bei denen die oben angesprochenen Aspekte eine erheblich bedeutendere Rolle spielen, gehört die Parallelisierung ereignisgesteuerter Simulatoren: Hierbei findet zwischen den einzelnen Simulatoren, die für jeweils einen Teil des Simulationsmodell zuständig sind und autonom, also ohne zentrale Steuerung, miteinander kooperieren, sowohl Synchronisation als auch Kommunikation häufig und unregelmäßig statt und es muß bei der Simulation von Ereignissen eine vorgegebene

---

"When a long series of identical computations is to be performed, such as those required for the formation of numerical tables, the machine can be brought into play so as to give several results at the same time, which will greatly abridge the whole amout of processes."

Ein frühes Beispiel für einen funktionsfähigen Parallelrechner ist das in [CW67] beschriebene Projekt, in dem ein SIMD-Rechner für ein parallel implementiertes Wetterprognose-System konstruiert wurde.

Reihenfolge eingehalten werden, die sich nur global, d. h. nicht für die Ereignisse jedes einzelnen Simulators separat definieren läßt. Da ereignisgesteuerte Simulation darüber hinaus ein sehr rechenzeitintensives Problem mit großer Praxisrelevanz ist, gibt es schon seit über zehn Jahren Versuche, ereignisgesteuerte Simulatoren zu parallelisieren. Auch hier hat die bisher geleistete Forschungsarbeit eher theoretisch-konzeptionellen Charakter, so daß die Frage, ob Parallelisierung bei praxisrelevanten Modellen tatsächlich ein geeignetes Mittel zur Leistungssteigerung ist, bis heute nicht beantwortet wurde.

Im Rahmen dieser Arbeit wird die Parallelisierung ereignisgesteuerter Simulatoren für Warteschlangennetze behandelt. Die Grundidee dabei ist, die Parallelisierung ereignisgesteuerter Simulatoren anhand einer konkreten Beispielklasse zu untersuchen, an der auch anwendungsspezifische Optimierungen durchführbar sind. Warteschlangennetze sind dabei im Grunde Stellvertreter für eine ganze Reihe ähnlicher Simulationsmodelle (Materialflußsysteme, Rechnernetze mit Paketvermittlung, ATM-Schaltnetzwerke, etc.). Die Untersuchungen werden ausschließlich auf nachrichtenbasierten MIMD-Parallelrechnern[2] durchgeführt, da nur dieser Rechnertyp für praktische Experimente zur Verfügung stand und gegenwärtig mit Vertretern wie der CM5, den Intel-Hypercube-Serien (iPSC, Paragon) sowie transputerbasierten Systemen relativ verbreitet ist.

Dabei werden in der vorliegenden Arbeit nicht nur die durch unterschiedliche Parallelisierungsprinzipien erzielbaren Leistungssteigerungen analysiert, sondern es wird vor allen Dingen auch der Frage nachgegangen, welche Probleme bei der Parallelisierung einer Leistungssteigerung im Wege stehen und eine Gewichtung dieser Probleme vorgenommen. Weiterhin wird die Frage untersucht, inwieweit es möglich ist, mit Methoden wie etwa der Kritischen-Pfad-Analyse die maximal mögliche Effizienz eines verteilten Simulators anhand von Laufzeit-Messungen am sequentiellen Simulator vorherzusagen.

Da diese Untersuchungen vielfach nur in geringem Maße durch spezielle Eigenschaften der parallelen Simulation gekennzeichnet sind, liefern die Ergebnisse dieser Arbeit auch einen Beitrag zu allgemeineren Problemen der Programmierung paralleler Systeme: Da die verwendeten Analysemethoden (Auswerten von Trace-Files und Kritische-Pfad-Analyse) auch schon für die Leistungsanalyse anderer paralleler Systeme vorgeschlagen wurden, sollten die hier thematisierten Aspekte auch auf andere Anwendungen übertragbar sein. Weiterhin hat die parallele ereignisgesteuerte Simulation Modellcharakter für Anwendungen mit feinkörniger Granularität. Die in dieser Arbeit dargestellten prinzipiellen und praktischen Schwierigkeiten bei

[2] Eine gute Beschreibung der unterschiedlichen Parallelrechner-Architekturen findet man in [Dun90].

der Realisierung eines effizienten Systems werden daher sicherlich auch bei anderen feingranularen Anwendungen auftauchen. Schließlich wird in dieser Arbeit auch auf Probleme bzgl. Effizienz und Zuverlässigkeit eingegangen, die bei der praktischen Arbeit mit den verwendeten Parallelrechnern auftraten[3].

Die Erkenntnisse dieser Arbeit lassen sich wie folgt zusammenfassen:

- Simulation auf einem Parallelrechner kann strukturelle Änderungen am Simulationsmodell erforderlich machen, die bereits zu einem deutlichen Effizienzverlust der Simulation des angepaßten Modells mit einem sequentiellen Rechner führen. Notwendigerweise hat daher ein Simulator auf einem Parallelrechner im allgemeinen mehr Simulationsarbeit zu leisten als ein Simulator auf einem sequentiellen Rechner. Dies wirkt sich natürlich negativ auf eine erzielbare Beschleunigung aus.

- Bedingt durch die besonderen Gegebenheiten eines verteilten Laufzeit- bzw. Betriebssystems ist selbst die Ausführung derjenigen Teile des Simulationssystems, die beim parallelen und beim sequentiellen Simulator identisch sind, auf einem Parallelrechner mit erheblich höherem Aufwand verbunden.

- Verteilte Algorithmen für die Synchronisation der Simulatoren untereinander und für die Berechnung globaler Größen sind zwar notwendig, wurden jedoch als Gegenstand der Forschung bisher eher überschätzt, da für ihre Effizienz im wesentlichen der Grundsatz "Je primitiver, desto besser!" gilt und Optimierungen an ihnen gegenüber Optimierungen an den bereits erwähnten Problemen ohnehin eher geringen Erfolg versprechen.

- Die mit Hilfe der Kritischen-Pfad-Analyse prognostizierten Beschleunigungen können nur grobe Anhaltspunkte für tatsächlich erreichbare Werte liefern, unter Umständen sind mit parallelen Simulatoren sogar Beschleunigungen erzielbar, die höher als die mit Kritischer-Pfad-Analyse berechneten Werte sind. Dagegen ist Kritische-Pfad-Analyse ein wirkungsvolles Hilfsmittel beim Erkennen von "inhärent sequentiellen" Simulationsmodellen, d. h. bei Modellen, bei denen eine effektive parallele Simulation von vornherein ausgeschlossen ist.

Insgesamt kann festgestellt werden, daß mit paralleler, ereignisgesteuerter Simulation bei sehr sorgfältiger Implementierung und bei Verfügbarkeit effizienter System-

[3] In der Literatur werden die Begriffe "parallel" und "verteilt" in zum Teil sehr unterschiedlichen Bedeutungen gebraucht. In dieser Arbeit wird ausschließlich der Begriff "Parallelrechner" bzw. "parallele, ereignisgesteuerte Simulation" benutzt, womit grundsätzlich ein nachrichtenbasierter MIMD-Rechner bzw. ein Simulationssystem darauf gemeint ist. Im Regelfall sind die betreffenden Aussagen jedoch auch für andere Klassen von MIMD-Rechnern (z. B. Shared-Memory-Systeme) und Rechnernetze gültig.

software mäßige Beschleunigungen möglich sind, sofern das jeweilige Simulationsmodell gewisse gutartige Eigenschaften aufweist und eine ausreichende Zahl von Prozessoren (typischerweise ≥ 8) zur Verfügung steht. Da die Realisierung solcher Systeme auf den augenblicklich verfügbaren Parallelrechnern ein schwieriges Unterfangen darstellt, ist allerdings nicht zu erwarten, daß sich in absehbarer Zeit parallele, ereignisgesteuerte Simulation auf breiter Front bei den Anwendern durchsetzen wird. Es gibt jedoch Spezialfälle (z. B. bei der Koppelung räumlich verteilter Simulationssysteme), bei denen sich der Einsatz dieser Methoden lohnt. Darüber hinaus zeigt sich einmal mehr, daß feingranulare Anwendungen schwer zu parallelisieren sind, da keine aufwendigen Synchronisationsalgorithmen verwendet werden dürfen und hohe Effizienzforderungen an die verwendete Systemsoftware gestellt werden müssen.

Die Arbeit gliedert sich in folgende Kapitel:

Kapitel 2 beschäftigt sich sowohl anschaulich als auch formal mit dem Begriff des Simulationsmodells. Dabei wird ausgehend vom Begriff des "sequentiellen Simulationsmodells", also eines Modells, das für die Simulation auf einem sequentiellen Rechner geeignet ist, der Begriff des "verteilten Simulationsmodells" entwickelt, das im wesentlichen aus einer Menge von miteinander kooperierenden sequentiellen Simulationsmodellen besteht und somit für die Simulation auf einem Parallelrechner geeignet ist.

Kapitel 3 enthält eine Einführung in das Gebiet der parallelen, ereignisgesteuerten Simulation sowie eine Zusammenfassung der aktuellen Forschungsergebnisse.

In Kapitel 4 wird die in dieser Arbeit untersuchte Klasse von Simulationsmodellen, nämlich Warteschlangennetze, beschrieben und die für die praktischen Untersuchungen verwendete Beschreibungssprache LAVENDER vorgestellt.

Kapitel 5 enthält die Beschreibung der Komponenten des Testbetts DISQUE, das für die in dieser Arbeit durchgeführten Untersuchungen erstellt wurde. Es wird dabei auf die verwendete Hardware, die verschieden instrumentierten sequentiellen und parallelen Simulatoren sowie die verwendeten Analysewerkzeuge eingegangen.

In Kapitel 6 werden die mit DISQUE durchgeführten Messungen beschrieben sowie die dabei beobachteten Phänomene erklärt. Dazu werden zunächst die simulierten Warteschlangennetze vorgestellt. Danach wird untersucht, welches die entscheidenden Probleme bei der Parallelisierung ereignisgesteuerter Simulatoren sind.

In Kapitel 7 werden die Ergebnisse aus Kapitel 6 zueinander in Beziehung gesetzt, um so zu einem Gesamtbild der Probleme und Möglichkeiten bei der Parallelisierung

ereignisgesteuerter Simulatoren zu kommen.

Kapitel 8 enthält Messungen und Analysen, die einen ersten Einblick in die Probleme und Möglichkeiten optimistischer Synchronisationsverfahren geben sollen.

In Kapitel 9 werden schließlich die Resultate dieser Arbeit sowie daraus resultierende allgemeine Erkenntnisse zusammengefaßt und offene Fragen sowie daraus ableitbare zukünftige Forschungsziele angesprochen.

Kapitel 2

Ereignisgesteuerte Simulationsmodelle

Laut VDI-Richtlinie 3663 ist Simulation die "Nachbildung eines dynamischen Prozesses in einem Modell, um zu Erkenntnissen zu gelangen, die auf die Wirklichkeit übertragbar sind". Grundlage einer Simulation ist also das Simulationsmodell, in dem die reale Welt abstrakt beschrieben wird. Dies geschieht im Regelfall unter bewußter Auslassung bestimmter Details des realen Systems, teils, weil sie für die Erkenntnisse, die man aus dem Simulationssystem gewinnen will, irrelevant sind, teils weil man schlicht nicht in der Lage ist, diese Details exakt zu beschreiben und hofft, daß ihre Auswirkungen auf die erstrebten Erkenntnisse eher gering sind.

Im Laufe der Zeit wurde eine Reihe ganz unterschiedlicher Konzepte entwickelt, mit denen man ein reales System für die Simulation modellieren kann (vgl. z. B. die Übersicht in [MM89]). Das in dieser Arbeit verwendete Konzept ist unter dem Namen "ereignisgesteuertes Simulationsmodell" bekannt. Die Grundidee dabei ist, das reale System durch eine Serie von Zustandsänderungen zu beschreiben, die atomar, d. h. ohne zeitliche Dauer sind. Diese Zustandsänderungen werden Ereignisse genannt, wobei ein Ereignis nicht nur den Zustand des Systems ändert, sondern außerdem auch weitere Ereignisse in der Zukunft bewirkt. (So kann beispielsweise bei einer Straßenverkehrssimulation das Ereignis "Auto verläßt Kreuzung" das Ereignis "Auto kommt an nächster Kreuzung an" bewirken.) Ereignisse brauchen dabei lediglich einen Teil des gesamten Systemzustands verändern, die Änderung des Gesamtzustands ergibt sich dann aus der Summe der Zustandsänderungen der Ereignisse. Diese Art der Simulation eignet sich besonders gut für die Modellierung realer Systeme, bei denen man sich mehr für Systemzustände als für Zustandsübergänge interessiert und bei denen die Beschreibung einzelner Komponenten eines Systems

leicht, die Beschreibung des gesamten Systemverhaltens aber nur schwer möglich ist (man denke z. B. an das Verhalten eines einzelnen Autos und den Autoverkehr einer Großstadt).

In diesem Kapitel geht es im wesentlichen um eine mathematisch exakte Definition des Begriffs "ereignisgesteuertes Simulationsmodell" sowie um eine Erweiterung dieses Begriffs für parallele, ereignisgesteuerte Simulationen. Dazu wird zunächst anschaulich die Arbeitsweise eines ereignisgesteuerten Simulators geschildert. Die Informationen, die ein solcher Simulator dafür benötigt, bilden dann gerade das ereignisgesteuerte Simulationsmodell. Weiterhin wird aus der Erkenntnis heraus, daß ein paralleler, ereignisgesteuerter Simulator aus einer Menge kommunizierender sequentieller Simulatoren besteht, der Begriff des Simulationsmodells für parallele, ereignisgesteuerte Simulation erweitert[1].

Die Definitionen dieses Kapitels dienen mehreren Zwecken:

- In der Literatur und im Hauptteil dieser Arbeit häufig verwendete Begriffe wie etwa "Ereignis" oder "Verschmelzen von Simulationsmodellen" lassen sich exakt definieren.
- Die speziellen Anforderungen an ein Simulationsmodell, die gewisse verteilte Simulationsalgorithmen stellen, sind mit ihrer Hilfe präzise formulierbar.
- Für existierende sequentielle und verteilte Simulationssprachen (wie etwa die in [Meh94] beschrieben Sprache DSL) können die Definitionen als Basis für eine operationale Semantik dieser Sprachen dienen.

Weiterhin wird der hier entwickelte Ansatz mit den Arbeiten von Misra [Mis86] und Pohlmann [Poh91] sowie den Arbeiten von Zeigler [Zei84] über prozeßorientierte Simulationsmodelle verglichen.

2.1 Sequentielle Simulationsmodelle

Um zu verstehen, wie man zu dem weiter unten beschriebenen mathematischen Modell kommt, betrachte man die Arbeitsweise eines auf einer sequentiellen Maschine laufenden *Simulators für ereignisgesteuerte Simulation* (Abb. 2.1). Er besteht im

[1] Dies zeigt, daß es sich bei der parallelen, ereignisgesteuerten Simulation eben nicht um "an entirely new approach to the problem of system simulation" handelt, wie in [Mis86] behauptet wird, sondern um eine (zugegebenermaßen nicht-triviale) Erweiterung existierender Konzepte.

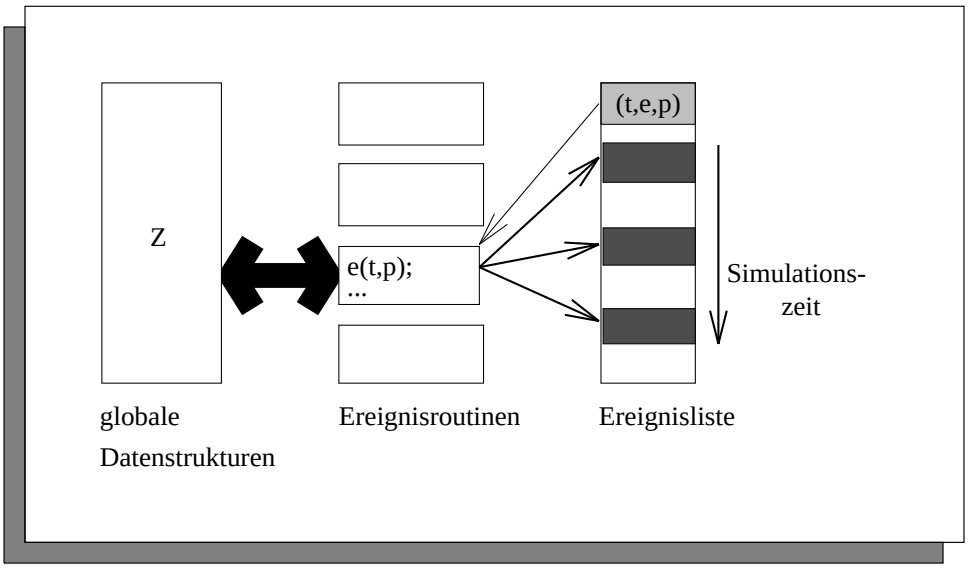


Abbildung 2.1: Sequentieller Simulator

wesentlichen aus einer Menge von Prozeduren, den sogenannten *Ereignisroutinen*, und einer Menge von globalen Datenstrukturen, deren Werte den Zustand des zugrundeliegenden realen Systems beschreiben. Der Aufruf einer solchen Ereignisroutine entspricht dem Eintreten eines Ereignisses in der Realität. Innerhalb der Ereignisroutine werden dann die globalen Datenstrukturen des Simulators manipuliert, um so den Effekt des realen Ereignisses im Simulationsmodell nachzubilden. Darüber hinaus werden im allgemeinen in einer solchen Ereignisroutine weitere Aufrufe von Ereignisroutinen veranlaßt. Diese Aufrufe sind mit *Zeitstempeln* versehen, die die Eintrittszeitpunkte der Ereignisse im realen System darstellen. Dies entspricht der Tatsache, daß in der Realität ein Ereignis weitere Ereignisse in der Zukunft zur Folge haben kann. (So kann beispielsweise in einer Straßenverkehrssimulation das Ereignis "Auto verläßt Kreuzung" das Ereignis "Auto kommt an Kreuzung an" für die Ankunftszeit des Autos an der nächsten Kreuzung veranlassen.) Alle auf diese Art erzeugten Aufrufe werden in einer zentralen Datenstruktur, der *Ereignisliste*, zwischengespeichert. Der Simulator arbeitet dann nach folgendem Verfahren:

```
while Ereignisliste nicht leer
do
 hole Aufruf mit kleinstem Zeitstempel aus der Ereignisliste;
 führe die entsprechenden Ereignisroutine aus;
 füge die dabei erzeugten Aufrufe in die Ereignisliste ein;
od
```

Das Verhalten eines sequentiellen Simulators ist also im wesentlichen durch die Semantik der Ereignisroutinen gekennzeichnet, wobei sowohl die von ihnen durchgeführten Manipulationen an den globalen Datenstrukturen, als auch die von ihnen neu erzeugten Ereignisse von Interesse sind. Dies führt dann zu folgender Definition:

Defininition 1
Ein **sequentielles, ereignisteuertes Simulationsmodell** $\mathcal{S}$ *ist ein 9-Tupel*

$$(Z, E, P, z_s, I, \varphi, \tau, \rho, \pi)$$

wobei

Z	*die (Abstraktion der) Menge der Zustände des simulierten Systems*
E	*die Menge der Bezeichner für Ereignisroutinen*
P	*die Menge der Parameterwerte für die Ereignisroutinen*
$z_s \in Z$	*der Startzustand*
$I \in \mathcal{P}_f(\mathbb{R} \times E \times P)$	*die Multimenge*[1] *der Initialereignisse*
$\varphi : \mathbb{R} \times E \times P \times Z \to Z$	*die Zustandstransformationen durch Ereignisroutinen*
$\tau : \mathbb{R} \times E \times P \times Z \to \mathcal{P}_f(\mathbb{R} \times E \times P)$	*die von Ereignisroutinen erzeugten Ereignisse*
$\rho : \mathbb{R} \times E \times P \times Z \to \mathcal{P}_f(\mathbb{R} \times E \times P)$	*die von Ereignisroutinen gelöschten Ereignisse*
$\pi : \mathbb{R} \times \mathcal{P}_f(E \times P) \to E \times P$	*die Auswahlfunktion*

beschreibt und zusätzlich gelten muß

(1) $\forall (t, e, p, z) \in \mathbb{R} \times E \times P \times Z : (t', e', p') \in \tau(t, e, p, z) \;\Rightarrow\; t' \geq t$

(2) $\forall (t, e, p, z) \in \mathbb{R} \times E \times P \times Z : (t', e', p') \in \rho(t, e, p, z) \;\Rightarrow\; t' \geq t$

(3) $\forall t \in \mathbb{R}\, \forall M \in \mathcal{P}_f(E \times P) : \pi(t, M) \in M$.

[1] Anschaulich formuliert ist eine endliche Multimenge eine endliche Menge, in der Elemente mehrfach vorkommen dürfen. Die Mengenoperationen wie Vereinigung, Durchschnitt, etc. sind entsprechend definiert. Für eine Menge M bezeichne $\mathcal{P}_f(M)$ die Menge aller endlichen Multimengen von M.

Ein Tripel $(t, e, p) \in \mathbb{R} \times E \times P$ *heißt* **Ereignis** *in* $\mathcal{S}$.

Die einzelnen Komponenten von $\mathcal{S}$ haben dabei die folgende Bedeutung:

- Z ist die Menge der möglichen Werte der Datenstrukturen des Simulators, z_s ist der Wert dieser Datenstrukturen vor Start des Simulationslaufs.

- Ein Ereignis, d. h. ein Tripel $(t, e, p) \in \mathbb{R} \times E \times P$, stellt einen Eintrag in der Ereignisliste des Simulators dar, und zwar den mit dem Zeitstempel t versehenen Aufruf der Ereignisroutine e, bei dem die Parameter p an e übergeben werden. Aus Gründen der einfacheren Darstellung wurde für alle Ereignisroutinen nur ein einziger Parameter und eine gemeinsame Menge von Parameterwerten angenommen, p kann also beispielsweise ein Tupel von Parametern für e sein.

- I ist der Inhalt der Ereignisliste vor dem Start des Simulationslaufs. Da in Ereignislisten grundsätzlich dasselbe Ereignis mehrfach auftreten kann und die Anzahl der identischen Ereignisse angibt, wie oft die entsprechende Ereignisroutine aufgerufen werden muß, hat I statt einer Mengen- eine Multimengen-Struktur.

- Die Funktionen φ, τ und ρ beschreiben das Verhalten der Ereignisroutinen. Arbeitet der Simulator im Zustand $z \in Z$ das Ereignis $(t, e, p) \in \mathbb{R} \times E \times P$ ab, so ist sein interner Zustand anschließend $\varphi(t, e, p, z)$. Insbesondere hat also die Ereignisroutine e außer auf den Parameter p und auf den internen Zustand z noch auf den Zeitstempel t Zugriff. Die Multimenge der von der Ereignisroutine e in diesem Fall neu erzeugten Ereignisse ist $\tau(t, e, p, z)$. Da hier die Möglichkeit bestehen sollte, ein Ereignis mehrfach zu erzeugen, wurde wiederum für $\tau(t, e, p, z)$ die Multimengen-Struktur gewählt. Das gleiche gilt für die von e aus der Ereignisliste gelöschten Ereignisse $\rho(t, e, p, z)$. Man beachte, daß τ und ρ stets endliche Multimengen erzeugen, was für Definition 2 (s. u.) benötigt wird und in der Praxis ohnehin keine Einschränkung darstellt.

- Die Funktion π dient dazu, aus einer Ereignisliste, bei der es mehr als ein Ereignis mit minimalem Zeitstempel gibt, dasjenige auszuwählen, welches als nächstes durch die entsprechende Ereignisroutine ausgeführt werden soll. Sie dient einerseits dazu, Nicht-Determinismus bei der Ausführung der Simulation durch den Simulator zu vermeiden, kann andererseits aber auch wichtig für die korrekte Modellierung des zu simulierenden Systems sein, z. B. bei gleichzeitiger Ankunft zweier Signale an einem VLSI-Baustein. Der gemeinsame Zeitstempel ist dann der erste Parameter von π, der zweite Parameter ist

die Multimenge der Ereignisse ohne den Zeitstempel. Bedingung (3) besagt, daß π in Abhängigkeit vom Zeitstempel ein Element aus dieser Multimenge auswählt, insbesondere gilt daher $\pi(t, \{(e,p)\}) = (e,p)$.

- Die Bedingungen (1) und (2) besagen, daß keine Ereignisroutine Ereignisse in ihrer Vergangenheit (d. h. also mit einem Zeitstempel, der kleiner ist als der des Ereignisses, daß sie gerade ausführt) einplanen oder löschen kann. Genau wie in der Realität kann bei einem ereignisgesteuerten Simulationsmodell also immer nur die Zukunft, nicht aber die Vergangenheit beeinflußt werden.

Ein sequentielles Simulationsmodell ist das, was der Anwender eines Simulators erstellt, wenn er das zu simulierende System modelliert. Die Aufgabe des Simulators ist somit nur, die entsprechenden Ereignisroutinen in der richtigen Reihenfolge mit den entsprechenden Parametern versehen aufzurufen. Diese Reihenfolge wird nun als nächstes exakt definiert. Sie spiegelt die bereits weiter oben beschriebene operationale Arbeitsweise eines Simulators wider.

Defininition 2
Sei $\mathcal{S} = (Z, E, P, z_s, I, \varphi, \tau, \rho, \pi)$ ein sequentielles Simulationsmodell mit $I \neq \emptyset$. Dann heißt die (endliche oder unendliche) Folge

$$(t_i, e_i, p_i)_{i \in \mathbb{N}},\ (t_i, e_i, p_i) \in \mathbb{R} \times E \times P$$

Ereignisfolge *von $\mathcal{S}$, falls zwei Folgen $(z_i)_{i \in \mathbb{N}}$, $z_i \in Z$ und $(M_i)_{i \in \mathbb{N}}$, $M_i \subset \mathbb{R} \times E \times P$, M_i Multimenge, gleicher Länge existieren und das folgende gilt:*

1.) $z_0 = z_s$ und $M_0 = I$. Für das Folgenglied (t_0, e_0, p_0) der Ereignisfolge gilt:

$$t_0 = \min\{t \in \mathbb{R} | (t, e, p) \in I\} \qquad (*)$$

$$(e_0, p_0) = \pi(t_0, \{(e,p) | (t_0, e, p) \in I\})$$

2.) Für das Folgenglied (t_i, e_i, p_i) sei

$$z_{i+1} = \varphi(t_i, e_i, p_i, z_i)$$

und

$$M_{i+1} = ((M_i \setminus \{(t_i, e_i, p_i)\}) \cup \tau(t_i, e_i, p_i, z_i)) \setminus \rho(t_i, e_i, p_i, z_i)$$

Ist $M_{i+1} = \emptyset$, so ist (t_i, e_i, p_i, z_i) das letzte Folgenglied der Ereignisfolge von S (d. h. die Simulation terminiert nach Abarbeitung des i-ten Ereignisses), andernfalls ist

$$t_{i+1} = \min\{t \in \mathbb{R} | (t, e, p) \in M_{i+1}\} \qquad (**)$$

$$(e_{i+1}, p_{i+1}) = \pi(t_{i+1}, \{(e,p) | (t_{i+1}, e, p) \in M_{i+1}\}).$$

Zu dieser Definition ist noch folgendes zu bemerken:

- Es läßt sich induktiv unmittelbar zeigen, daß immer $|M_i| < \infty$ gilt, $(^{**})$ ist somit wohldefiniert. M_i entspricht dem Inhalt der Ereignisliste vor dem Abarbeiten des i-ten Ereignisses der Ereignisfolge.

- Wird in den Gleichungen $(*)$ und $(**)$ das Minimum mehrfach angenommen, wird mit Hilfe der Funktion π das Ereignis ausgewählt, das als nächstes abgearbeitet werden soll. Dabei kann ein überraschendes Phänomen auftreten: Erzeugt das ausgewählte Ereignis wiederum ein Ereignis mit demselben Zeitstempel, so ist es durchaus möglich, daß das neu erzeugte Ereignis bzgl. der Auswahlfunktion π "höhere Priorität" hat als das erzeugende Ereignis. Da jedoch in der Ereignisfolge die erzeugenden Ereignisse stets vor den erzeugten Ereignissen kommen, tritt es dort erst nach seinem erzeugenden Ereignis auf.

- Obwohl die Folge $(M_i)_{i\in\mathbb{N}}$ aus endlichen Multimengen besteht, kann trotzdem für unendliche Ereignisfolgen $((t_i, e_i, p_i))_{i\in\mathbb{N}}$ die Folge der Zeitpunkte $(t_i)_{i\in\mathbb{N}}$ beschränkt sein. Dies ist z. B. bei einem Simulationsmodell der Fall, bei dem am Anfang nur zwei Ereignisse mit den Zeitstempeln 1 und 2 vorhanden sind und bei dem bei der Ausführung eines Ereignisses mit Zeitstempel t ein neues Ereignis mit dem Zeitstempel $t/2 + 1$ erzeugt wird. Dies führt insbesondere dazu, daß das Ereignis mit dem Zeitstempel 2 nicht zur Ereignisfolge gehört und der Zeitpunkt 2 im Simulationsmodell somit nie auftritt, da anschaulich gesprochen die Ereignisse sich kurz vor diesem Zeitpunkt überstürzen. Der Modellierer kann also auf diese Art "Endlosschleifen" modellieren!

2.2 Verteilte Simulationsmodelle

Will man den im vorherigen Abschnitt skizzierten sequentiellen Simulator parallelisieren, so ergeben sich zunächst zwei grundsätzliche Möglichkeiten: Einmal ist es möglich, Teilkomponenten des Simulators, wie etwa den Zugriff auf die Ereignisliste, auf mehrere Prozessoren auszulagern, zum anderen besteht die Möglichkeit, mehrere Ereignisse (d. h. die Prozeduren, die die Ereignisroutinen repräsentieren) parallel auszuführen.

Die erstere Methode hat bereits den generellen Nachteil, daß es in einem Simulator ohnehin nur wenige Teilkomponenten gibt und somit der auf diese Art vorhandene Parallelismus von vornherein stark beschränkt ist. Weiterhin ergaben sich bei praktischen Versuchen mit diesem Parallelisierungsansatz bedingt durch die starken

Abhängigkeiten zwischen den Teilkomponenten keine nennenswerte Beschleunigungen (vgl. etwa [JCRB89]).

Will man mehrere Ereignisse gleichzeitig ausführen, so stößt man sofort auf das Problem, daß die Ausführung eines Ereignisses die globalen Datenstrukturen verändert und somit prinzipiell die Ausführung aller (bzgl. der Ereignisfolge) nachfolgenden Ereignisse beeinflussen kann, ein gleichzeitiges Ausführen mehrerer Ereignisse also zunächst nicht möglich ist. Eine Lösung des Problems findet sich für viele Modelle darin, daß die Ereignisroutinen jeweils nur auf bestimmte, disjunkte Teile der globalen Datenstrukturen zugreifen. In diesem Fall kann man das gesamte Simulationsmodell in Teilmodelle zerlegen, wobei alle Ereignisroutinen, die auf dieselben Datenstrukturen zugreifen, in einem Teilmodell zusammengefaßt werden. Dabei werden jedoch bei der Ausführung von Ereignissen im Regelfall Ereignisse neu erzeugt, die zu einem anderen Teilmodell gehören, falls es sich nicht um den trivial parallelisierbaren Fall von völlig unabhängigen Teilmodellen handelt (Abb. 2.2).

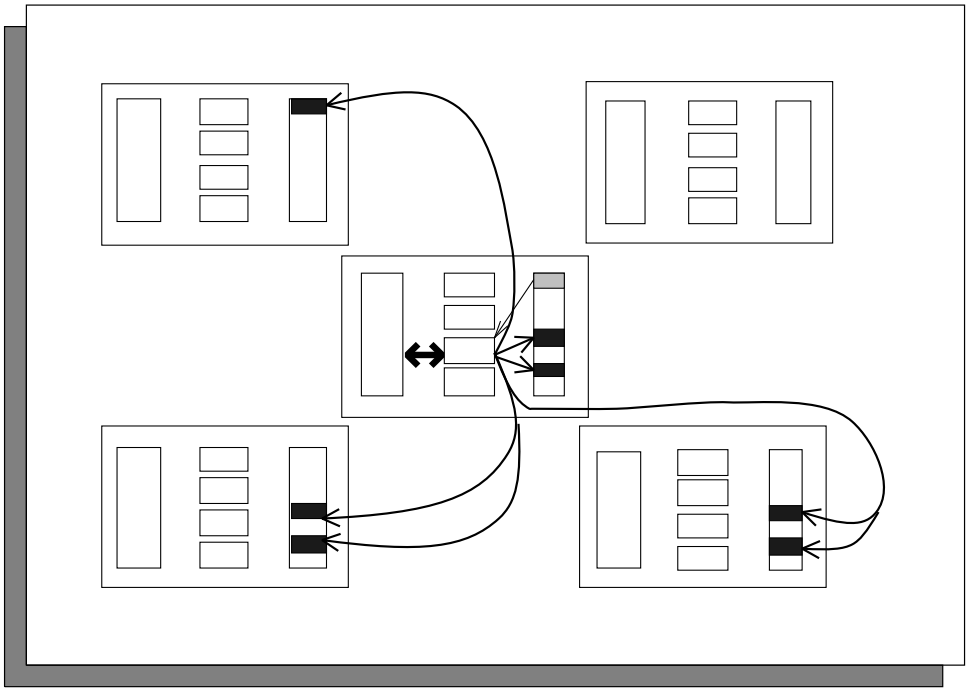


Abbildung 2.2: Verteilter Simulator aus fünf Teilmodellen

Diese Überlegungen führen dann zu folgender Definition:

Definition 3 *Für $n \in \mathbb{N}$ ist ein* **n-fach verteiltes Simulationsmodell** $\mathcal{S}_n$ *ein 9-Tupel*

$$((Z_1, \ldots, Z_n), (E_1, \ldots, E_n), P, (z_s^1, \ldots, z_s^n), I, (\varphi_1, \ldots, \varphi_n), (\tau_1, \ldots, \tau_n), (\rho_1, \ldots, \rho_n), \pi)$$

wobei

Z_i	*die Menge der Zustände des i-ten Teilmodells*
E_i	*die Menge der Bezeichner für Ereignisroutinen des i-ten Teilmodells*
P	*die Menge der Parameterwerte für die Ereignisroutinen*
$(z_s^1, \ldots, z_s^n) \in Z_1 \times \ldots \times Z_n$	*die Startzustände der einzelnen Teilmodelle*
$I \in \mathcal{P}_f(\mathbb{R} \times E \times P), \;\; E = \bigcup_{j=1}^n E_j$	*die Multimenge der Initialereignisse aller Teilmodelle*
$\varphi_i : \mathbb{R} \times E_i \times P \times Z_i \to Z_i$	*die lokalen Zustandstransformationen durch Ereignisroutinen des i-ten Teilmodells*
$\tau_i : \mathbb{R} \times E_i \times P \times Z_i \to \mathcal{P}_f(\mathbb{R} \times E \times P)$	*die von den Ereignisroutinen des i-ten Teilmodells erzeugten Ereignisse*
$\rho_i : \mathbb{R} \times E_i \times P \times Z_i \to \mathcal{P}_f(\mathbb{R} \times E \times P)$	*die von den Ereignisroutinen des i-ten Teilmodells gelöschten Ereignisse*
$\pi : \mathbb{R} \times \mathcal{P}_f(E \times P) \to E \times P$	*die globale Auswahlfunktion*

beschreibt und zusätzlich gelten muß

(1) $\forall i, j \in \{1, \ldots, n\} : i \neq j \;\Rightarrow\; E_i \cap E_j = \emptyset$

(2) $\forall i \in \{1, \ldots, n\}\; \forall (t, e, p, z) \in \mathbb{R} \times E_i \times P \times Z : (t', e', p') \in \tau_i(t, e, p, z) \;\Rightarrow\; t' \geq t$

(3) $\forall i \in \{1, \ldots, n\}\; \forall (t, e, p, z) \in \mathbb{R} \times E_i \times P \times Z : (t', e', p') \in \rho_i(t, e, p, z) \;\Rightarrow\; t' \geq t$

(4) $\forall t \in \mathbb{R}\; \forall M \in \mathcal{P}_f(E \times P) : \pi(t, M) \in M$.

Ein Tripel $(t, e, p) \in \mathbb{R} \times E \times P$ *heißt* **Ereignis** *in* $\mathcal{S}_n$.

Hierzu ist folgendes zu bemerken:

- Die 9-Tupel $(Z_i, E_i, P, z_s^i, \{(t, e, p) \in I | e \in E_i\}, \varphi_i, \tau_i, \rho_i, \pi)$ stellen sequentielle Simulationsmodelle dar, die außer sich selbst auch anderen Modellen Ereignisse einplanen bzw. löschen können. (Die Werte, die τ_i und ρ_i annehmen können, sind Multimengen über der Menge *aller* Ereignisse.) Die Ereignisroutinen der

einzelnen Teilmodelle können dabei nur auf ihren eigenen Zustand zugreifen, nicht auf die Zustände anderer Teilmodelle, wie man am Definitionsbereich der Funktionen τ_i, ρ_i und φ_i erkennt.

- Ein 1-fach verteiltes Simulationsmodell ist ein sequentielles Simulationsmodell.
- Die Auswahlfunktion muß hier global gewählt werden, da mehrere Ereignisse in den Ereignislisten der verschiedenen Simulatoren den kleinsten Zeitstempel des gesamten Systems tragen können. Ein Beispiel für eine solche Auswahlfunktion ist z. B. die Vergabe von Prioritäten für die Simulationsmodelle und für die Ereignisroutinen innerhalb der einzelnen Simulationsmodelle.
- Bedingung (1) wurde aus Gründen der einfacheren Darstellung hinzugenommen und besagt, daß die Bezeichner für Ereignisroutinen eindeutig innerhalb des Gesamtmodells gewählt werden müssen.

Analog zu Definition 2 ließe sich auch hier die Ereignisfolge eines verteilten Simulationsmodells definieren, doch wird dies auf den nächsten Abschnitt verschoben, da das dort eingeführte "Verschmelzen von Teilmodellen" diese Definition stark vereinfacht.

2.3 Verschmelzen von Teilmodellen

Bei einer Vielzahl der in der Praxis auftretenden verteilten Simulationsmodelle übersteigt die Anzahl der Teilmodelle bei weitem die Anzahl der Prozessoren des für die Simulation zur Verfügung stehenden Parallelrechners. Beispielsweise kann man bei VLSI-Schaltungen jedes Gatter als ein eigenes Teilmodell auffassen, was je nach Schaltung unter Umständen zu zehntausenden von Teilmodellen führen kann. Natürlich kann man sich auf den Standpunkt stellen, daß es in der Verantwortung des Modellierers liegt, das verteilte Simulationsmodell so zu gestalten, daß es die gewünschte Anzahl von Teilmodellen enthält. (Bei VLSI-Schaltungen wäre es z. B. möglich, größere Einheiten wie Speicher, ALU etc. jeweils als ein Teilmodell darzustellen.) Dies hätte jedoch den Nachteil, daß für jede Prozessorzahl ein neues Modell entworfen werden muß. Weiterhin wäre es damit Aufgabe des Modellierers, für eine gleichmäßige Verteilung der Rechenlast auf die einzelnen Knoten zu sorgen, was von der verwendeten Hardware, Compiler etc. abängig ist und daher eigentlich nichts mehr mit dem Simulationsmodell zu tun haben sollte.

Eine andere Möglichkeit besteht darin, durch Multithreading oder Multitasking mehrere Simulatoren quasi-parallel auf jeweils einem Prozessor ablaufen zu lassen. Dies ist eine durchaus praktikable Lösung, solange sich die Anzahl der Teilmodelle

in Grenzen hält; bei der Simulation eines VLSI-Bausteins mit zehntausenden von Gattern ist dies jedoch schon aus Gründen der Effizienz nicht mehr sinnvoll, da der Verwaltungsaufwand zu hoch wird.

Eine weitere Möglichkeit besteht darin, mehrere Teilmodelle durch mathematische Operationen zu einem einzigen neuen Teilmodell zusammenzufassen. Dies kann dann beim Start der Simulation automatisch geschehen, erfordert also keine Änderungen durch den Modellierer am Simulationsmodell. Die Idee dabei ist, die Datenstrukturen der entsprechenden Teilmodelle zusammenzufassen und die jeweilige Ereignisroutine nur auf dem Teil der Datenstruktur ablaufen zu lassen, der zu demjenigen Teilmodell gehört, von dem die Ereignisroutine stammt (Abb. 2.3). Dies führt

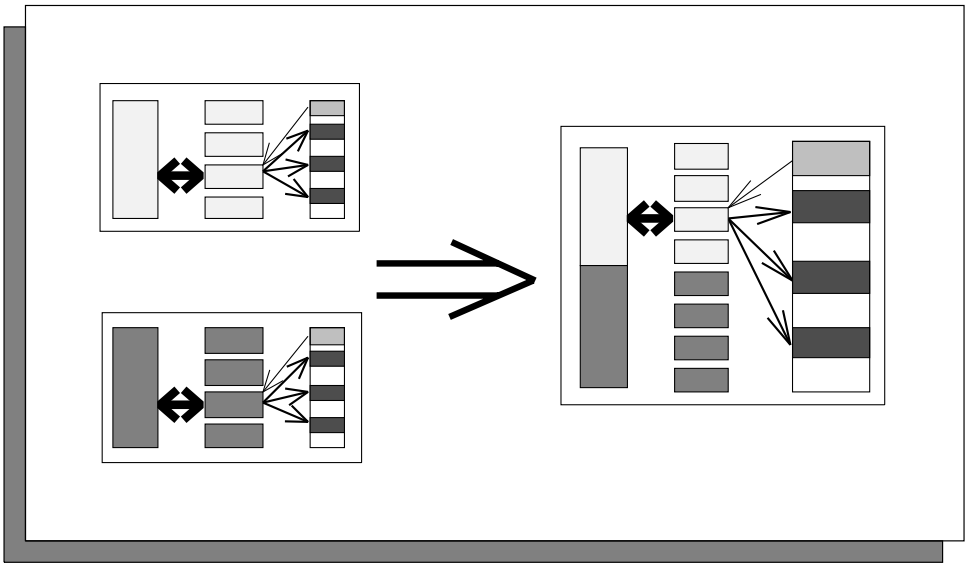


Abbildung 2.3: Verschmelzen von Teilmodellen

zu folgender Definition:

Defininition 4
Sei

$$\mathcal{S}_n = ((Z_1, \ldots, Z_n), (E_1, \ldots, E_n), P, (z_s^1, \ldots, z_s^n), I, (\varphi_1, \ldots, \varphi_n), (\tau_1, \ldots, \tau_n), (\rho_1, \ldots, \rho_n), \pi)$$

ein n-fach verteiltes Simulationsmodell, $k \in \mathbb{N}$, $k \leq n$ *und* $P = \{P_1, \ldots, P_k\}$ *eine Partition von* $\{1 \ldots n\}$ *(d. h.* $\bigcup_j P_j = \{1 \ldots n\}$, $\forall i \neq j : P_i \cap P_j = \emptyset$*). Dann ist das k-fach verteilte Simulationsmodell*

$$\mathcal{S}_n^P = ((\hat{Z}_1, \ldots, \hat{Z}_k), (\hat{E}_1, \ldots, \hat{E}_k), P, (\hat{z}_s^1, \ldots, \hat{z}_s^k), I, (\hat{\varphi}_1, \ldots, \hat{\varphi}_k), (\hat{\tau}_1, \ldots, \hat{\tau}_k), (\hat{\rho}_1, \ldots, \hat{\rho}_k), \pi)$$

wie folgt definiert:

Für $P_i = \{i_1, \ldots, i_l\}$ *sei*

$$\hat{Z}_i = Z_{i_1} \times \ldots \times Z_{i_l}$$

$$\hat{E}_i = E_{i_1} \cup \ldots \cup E_{i_l}$$

$$\hat{z}_s^i = (z_s^{i_1}, \ldots, z_s^{i_l})$$

$$\hat{\varphi}_i : \mathbb{R} \times \hat{E}_i \times P \times \hat{Z}_i \to \hat{Z}_i$$

$$\hat{\varphi}_i(t, e, p, (z_1, \ldots, z_l)) = (u_1, \ldots, u_l)$$

$$mit\ u_k = \begin{cases} \varphi_{i_k}(t, e, p, z_k) & falls\ e \in E_{i_k} \\ z_k & sonst \end{cases}$$

$$\hat{\tau}_i : \mathbb{R} \times \hat{E}_i \times P \times \hat{Z}_i \to \mathcal{P}_f(\mathbb{R} \times \bigcup_j \hat{E}_j \times P)$$

$$\hat{\tau}_i(t, e, p, (z_1, \ldots, z_l)) = \tau_{i_k}(t, e, p, z_k)\ mit\ e \in E_{i_k}\ \ (k\ ist\ eindeutig!)$$

$$\hat{\rho}_i : \mathbb{R} \times \hat{E}_i \times P \times \hat{Z}_i \to \mathcal{P}_f(\mathbb{R} \times \bigcup_j \hat{E}_j \times P)$$

$$\hat{\rho}_i(t, e, p, (z_1, \ldots, z_l)) = \rho_{i_k}(t, e, p, z_k)\ mit\ e \in E_{i_k}$$

Ein Sonderfall ist dabei $P = \{\{1, \ldots, n\}\}$. In diesem Fall sind alle Teilmodelle zu einem einzigen zusammengefaßt, man erhält also ein sequentielles Simulationsmodell. Dies kann man dazu benutzen, die Ereignisfolge eines verteilten Simulationsmodells zu definieren:

Defininition 5 *Sei* $\mathcal{S}_n$ *ein* n*-fach verteiltes Simulationsmodell und* $(t_i, e_i, p_i)_{i \in \mathbb{N}}$ *die Ereignisfolge von* $\mathcal{S}_n^{\{\{1,\ldots,n\}\}}$*. Dann heißt* $(t_i, e_i, p_i)_{i \in \mathbb{N}}$ *die Ereignisfolge von* $\mathcal{S}_n$*.*

Für $k \in \{1, \ldots, n\}$ *sei* $I_k := \{i | e_i \in E_k\}$*. Die Folge* $(t_i, e_i, p_i)_{i \in I_k}$ *heißt die Ereignisfolge des* k*-ten Teilmodells von* $\mathcal{S}_n$*.*

Um sich diesen Begriff zu veranschaulichen, denke man daran, daß ein sequentieller Simulator, der $\mathcal{S}_n^{\{\{1,\ldots,n\}\}}$ simuliert, immer das Ereignis mit dem kleinsten Zeitstempel seiner Ereignisliste ausführt. Man stelle sich nun einmal vor, er würde die Ereignisse in mehreren Ereignislisten, nach Teilmodellen getrennt verwalten. In diesem Fall würde er dann immer das Ereignis mit dem global kleinsten Zeitstempel aller Teilmodelle ausführen. Tatsächlich ist er aber an diese Reihenfolge nicht gebunden. Vielmehr darf er jedes Ereignis ausführen, das den kleinsten Zeitstempel der Ereignisliste seines Teilmodells trägt, wenn sichergestellt ist, daß nicht noch bei der Ausführung von Ereignissen anderer Teilmodelle ein Ereignis für sein Teilmodell entsteht, das einen kleineren Zeitstempel trägt. Anders formuliert ist er nur an die Reihenfolge der Ereignisse in den Ereignisfolgen seiner Teilmodelle, nicht aber an deren Reihenfolge in der Ereignisfolge des gesamten Modells gebunden. Somit können die Ereignisse der Ereignisfolgen der Teilmodelle von mehreren, parallel arbeitenden sequentiellen Simulatoren ausgeführt werden. Dabei müssen die Simulatoren jedoch u. U. solange mit der Ausführung eines Ereignisses warten, bis sichergestellt ist, daß es sich um das nächste Ereignis ihrer Ereignisfolge handelt. Dies ist das zentrale Problem bei der parallelen, ereignisgesteuerten Simulation. Die verschiedenen Ansätze dazu werden in Kapitel 3 beschrieben.

2.4 Vergleich des Modellbegriffs mit anderen Arbeiten

Beim Vergleich des eben geschilderten Modellbegriffs mit anderen Arbeiten ist generell zu bemerken, daß in den meisten Arbeiten über verteilte Simulation der Begriff des Simulationsmodells ausschließlich informell benutzt wird, da dort effiziente Simulationsalgorithmen für Parallelrechner im Vordergrund stehen. Im folgenden wird auf die Arbeit von drei Autoren eingegangen, bei denen dieser Begriff formaler gefaßt wird.

In [Mis86] wird von Misra der Begriff des physischen Systems (physical system) eingeführt. Dieses besteht aus einer Menge von physischen Prozessen (physical processes), die sich gegenseitig Nachrichten zuschicken können. Dies geschieht in Nullzeit, so daß jeder Nachricht ein (Simulations-)Zeitpunkt zuordenbar ist. (Nachrichtenlaufzeiten werden durch Untätigkeitsphasen des entsprechenden physischen Prozesses beim Senden oder Empfangen der Nachricht modelliert.) An ein solches Modell werden zwei Anforderungen gestellt:

Realizability: Die Nachrichten, die ein physischer Prozeß zum Zeitpunkt t sendet, hängen nur von seinem Anfangszustand, von t und von den Nachrichten ab, die er

bis einschließlich t erhalten hat.

Predictability: Falls es einen Zyklus von physischen Prozessen gibt, d. h. physische Prozesse $pp_1, \ldots, pp_n$, bei denen pp_i an pp_{i+1} Nachrichten senden kann, muß es für jeden Zeitpunkt t und für jeden Zyklus einen physischen Prozeß des Zyklus und ein $\epsilon > 0$ geben, so daß aus den Nachrichten, die der physische Prozeß bis einschließlich t empfängt, die Nachrichten, die er bis $t + \epsilon$ entlang des Zyklus sendet, bestimmbar sind.

Diese zwei Bedingungen sollen eine Art Wohldefiniertheit garantieren, in dem Sinne, daß sämtliche Nachrichten, die innerhalb des physischen Systems bis zu einem gegebenen Zeitpunkt verschickt werden, aus dem Anfangszustand des Systems berechenbar sind. Ein großes Problem ist dabei, daß sämtliche Begriffe nur sehr informell definiert sind, so daß man eigentlich nicht von einem mathematischen Modell sprechen kann.

Pohlmann greift in [Poh91] die Ideen von Misra auf und formalisiert sie mit Hilfe des Berechnungsmodells von Broy [Bro82]. Ein Simulationsmodell besteht dort aus einem gerichteten Graphen, bei dem sich die Knoten entlang der Kanten Nachrichten (d. h. Ereignisse) zuschicken. Der Zeitbegriff ist diskretisiert, so daß das Verhalten einer Kante durch eine unendliche Folge von Nachrichten und "Ticks" (Zeichen für "Keine Nachricht zu diesem Zeitpunkt") beschrieben werden kann. Das Verhalten der Knoten wird durch Funktionen spezifiziert, die den Nachrichtenstrom einer aus einem Knoten herausführenden Kante als Funktion der Nachrichtenströme der in den Knoten hereinführenden Kanten beschreibt (Abb. 2.4). Führt man einen Vektor ein, der für den Nachrichtenstrom jeder Kante eine Variable enthält, so läßt sich aus diesen Funktionen eine Fixpunktgleichung für diesen Vektor aufstellen. Bei gewissen Zusatzbedingungen ähnlich denen von Misra ist dieser Fixpunkt eindeutig. Die Aufgabe eines Simulators ist es, den Fixpunkt zu berechnen. In [Poh91] werden einige bekannte Verfahren der parallelen, ereignisgesteuerten Simulation als spezielle Fixpunktiterationen dargestellt.

Das Hauptproblem mit den Modellen von Misra und Pohlmann ist die Tatsache, daß die Funktionen, die das Verhalten der physischen Prozesse bzw. Knoten beschreiben, von *allen* bis zum Zeitpunkt des zu simulierenden Ereignisses eingetroffenen Nachrichten abhängen. In allen gängigen Simulationssprachen und natürlich auch in mit konventionellen imperativen Sprachen implementierten Simulatoren hat man statt dessen Datenstrukturen, deren Werte nicht die Historie, sondern den Endzustand des bisher eingegangenen Nachrichtenstroms widerspiegeln. Dies ist auch schon aus Gründen des begrenzten Speicherplatzes sinnvoll. Ein Anwender, der verteilte Simulationsalgorithmen, die mit Hilfe der erwähnten Modelle formuliert sind, in Implementierungen umsetzen will, hat daher einiges an Arbeit zu leisten.

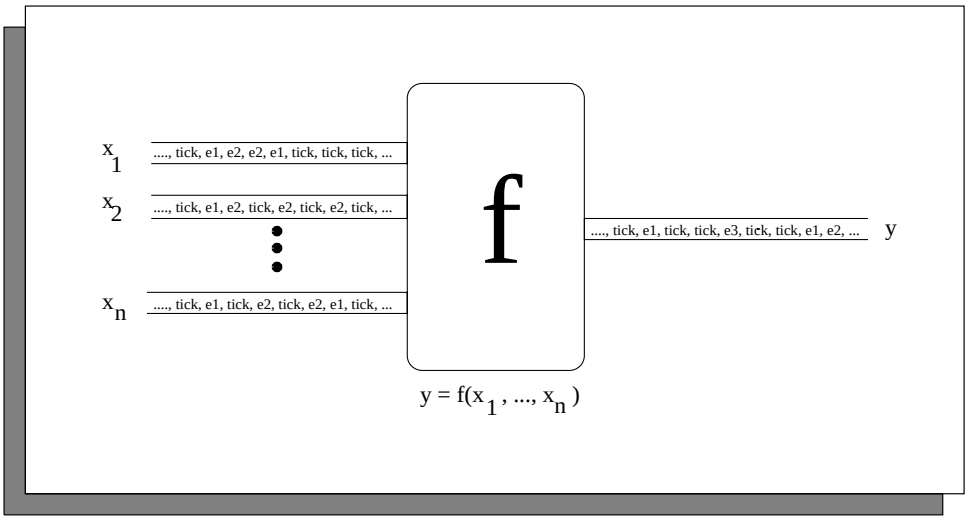


Abbildung 2.4: Pohlmanns verteiltes Simulationsmodell

Weiterhin ist in diesen Modellen keine Möglichkeit vorgesehen, Ereignisse wieder zu löschen. Dies ist in praktischen Anwendungen jedoch häufig notwendig. Ein Ausweg besteht dann nur darin, spezielle Lösch-Nachrichten zu schicken, die den Einfluß der zugehörigen Ereignisnachricht auf das Verhalten des entsprechenden Knoten annulieren. Zusammenfassend kann man sagen, daß sich die Modelle wohl eher für theoretische Betrachtungen eignen.

Eine größere Ähnlichkeit besteht zwischen den hier eingeführten Simultionsmodellen und den Arbeiten von Zeigler (eine umfassende Darstellung ist [Zei84]). Im Unterschied zu den hier entwickelten Begriffen handelt es sich dort jedoch um die prozeßorientierte Sicht der ereignisgesteuerten Simulation. Der zentrale Begriff dabei ist das DEVS (= **D**iscrete **EV**ent **S**ystem specification, vgl. Abb. 2.5). Ein DEVS besteht dabei aus einer Menge von Zuständen S mit Zustandsdauerfunktion ta, einer Menge von externen Ereignissen X, sowie zwei Übergangsfunktionen δ_ϕ und δ_{ex}. Befindet sich ein DEVS im Zustand $s \in S$, so verharrt es darin so lange, bis entweder die Zeit $ta(s)$ abgelaufen ist, oder ein externes Ereignis $x \in X$ eintrifft. Ist die Zeit von s abgelaufen, so geht das DEVS anschließend in den Zustand $\delta_\phi(s)$. Ist ein externes Ereignis x eingetroffen und ist seit dem Wechsel in den Zustand s die Zeit $e < ta(s)$ vergangen, so geht das DEVS in den Zustand $\delta_{ex}(s, e, x)$. In beiden Fällen beginnt die "Uhr" des neuen Zustands dann wieder von 0 an zu laufen.

Ein Anwender, der ein DEVS simulieren möchte, übergibt dem Simulator nicht nur die Beschreibung des DEVS, sondern auch einen Startzustand $s \in S$ mit bereits abgelaufener Zeit e und eine endliche Menge mit Zeitstempeln versehener externer Ereignisse $\{(x_i, t_i)\}$, wobei die t_i in einem Zeitintervall $[a, b]$ liegen. Der Simulator

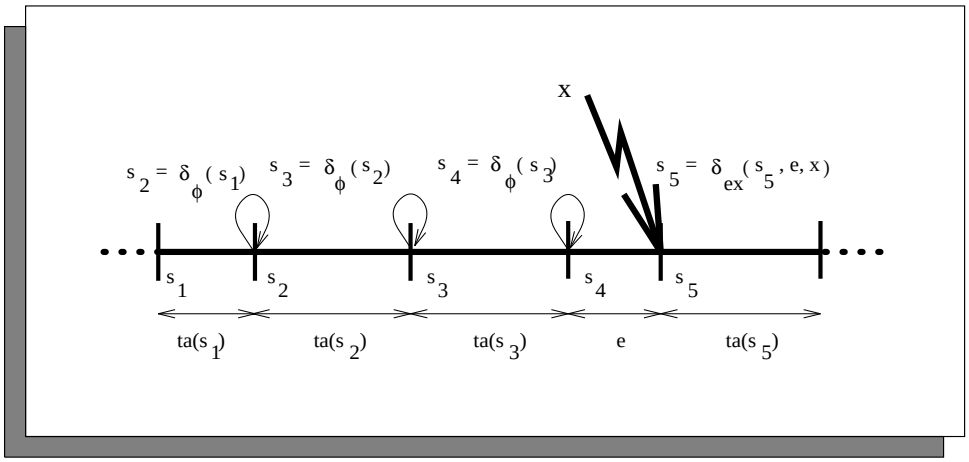


Abbildung 2.5: Zeiglers DEVS

berechnet dann den Verlauf der Zustände des DEVS im Intervall $[a, b]$, also eine Abbildung $[a, b] \rightarrow S$.

Vergleicht man dies mit hier eingeführten Begriff des sequentiellen Simulationsmodells, so ergeben sich deutliche Ähnlichkeiten: In beiden Fällen spezifiziert der Anwender einen Startzustand, Zustandsübergänge, die das System selbst erzeugt (erzeugte Ereignisse bzw. Zustandsübergänge) und "von außen" erzeugte Zustandsübergänge (initiale bzw. externe Ereignisse). Der Simulator errechnet daraus die Folge der tatsächlich durchgeführten Zustandsübergänge (Ereignisfolge bzw. Zustandsverlauf).

Das Analogon zu dem hier eingeführten Begriff des verteilten Simulationsmodells gibt es auch bei Zeigler, nämlich das "Multikomponenten-DEVS in modularer Form". Dieses besteht aus einer Menge von DEVS, bei denen nach Ablauf der Zeit eines Zustands zusätzlich zum Übergang in den neuen Zustand von dem entsprechenden DEVS externe Ereignisse für andere DEVS generiert werden, die diese wie alle anderen externen Ereignisse behandeln. Interessanterweise braucht auch Zeigler dabei eine Auswahlfunktion, die aus einer Menge von DEVS, bei denen die "Uhren" ihrer Zustände gleichzeitig abgelaufen sind, dasjenige DEVS auswählt, dessen Zustandsänderung zuerst ausgeführt wird.

Obwohl der von Zeigler benutzte Modellbegriff große Ähnlichkeit mit dem für diese Arbeit entwickelten Begriff hat, ist er doch für völlig andere Zwecke entwickelt worden, nämlich zur Beschreibung von hierarchisch strukturierten Simululationsmodellen. Das hier entwickelte Modell wurde dagegen gezielt als Grundlage für die Beschreibung der Voraussetzungen und der Arbeitsweise paralleler Synchronisationsverfahren entwickelt und wird entsprechend in Kapitel 3 eingesetzt.

Kapitel 3

Stand der Forschung in der parallelen, ereignisgesteuerten Simulation

In diesem Kapitel werden zunächst die grundsätzlichen Vorgehensweisen bei der parallelen Realisierung ereignisteuerter Simulatoren sowie die dabei auftretenden Probleme erläutert. Für diese Probleme gibt es zwei grundsätzliche Lösungsansätze, die im Anschluß daran geschildert werden. Weiterhin werden Methoden beschrieben, mit deren Hilfe man analysieren kann, wie gut sich ein verteiltes Simulationsmodell für parallele Simulation eignet. Als Abschluß dieses Kapitels wird eines der bekanntesten parallelen Simulationssysteme vorgestellt und auf die dort erzielten Geschwindigkeitsgewinne eingegangen. Da es bereits gute Übersichtsartikel zum Gebiet der parallelen, ereignisgesteuerten Simulation gibt (z. B. [Fuj90, RW89]), wurde dieses Kapitel bewußt knapp und übersichtlich gehalten, wobei im Gegensatz zu den bisher existierenden Übersichten der in Kapitel 2 entwickelte Formalismus als einheitliche Grundlage benutzt und der Schwerpunkt gemäß den später in dieser Arbeit untersuchten Aspekten gewählt wurde.

Im folgenden bezeichnet

$$\mathcal{S}_n = ((E_1, \ldots, E_n), P, (z_s^1, \ldots, z_s^n), I, (\varphi_1, \ldots, \varphi_n), (\tau_1, \ldots, \tau_n), (\rho_1, \ldots, \rho_n), \pi)$$

stets ein n-fach verteiltes Simulationsmodell und $(t_i, e_i, p_i)_{i \in \mathbb{N}}$ seine Ereignisfolge. Weiterhin bezeichne $(t_i^k, e_i^k, p_i^k)_{i \in \mathbb{N}}$, $k \in \mathbb{N}$ die Ereignisfolge des k-ten Simulationsmodells.

3.1 Das zugrundeliegende Parallelisierungskonzept

Wie in Kapitel 2.2 bereits erläutert, besteht die Grundidee der parallelen, ereignisgesteuerten Simulation darin, mehrere Ereignisroutinen gleichzeitig auf mehreren Knoten eines Parallelrechners auszuführen. Da verteilte, ereignisgesteuerte Simulationsmodelle entsprechend Definition 3 gerade so konstruiert wurden, daß Ereignisse verschiedener Teilmodelle auf disjunkten Datenstrukturen ausgeführt werden, wird dazu jedem Teilmodell ein eigener Simulator, logischer Prozeß (LP)[1] genannt, zugeordnet. Dieser ist im Prinzip genauso aufgebaut wie ein sequentieller Simulator (vgl. Abb. 2.2), enthält also neben den globalen Datenstrukturen und Ereignisroutinen auch eine eigene Ereignisliste. Wie am Ende von Kapitel 2.3 bereits ausgeführt wurde, kann er im Prinzip genauso wie ein sequentieller Simulator arbeiten, d. h. das jeweils vorderste Ereignis seiner Ereignisliste ausführen und ggf. neue Ereignisse erzeugen. Das zentrale Problem dabei ist jedoch, daß er dazu sicher sein muß, nicht noch von einem anderen LP ein Ereignis eingeplant zu bekommen, das in der Ereignisfolge seines Teilmodells vor dem vordersten Ereignis seiner Ereignisliste kommt, d. h. beim Eintreffen zum vordersten Ereignis seiner Ereignisliste würde. In der Literatur ist dies als "Time-Advancement-Problem" bekannt, präzise läßt es sich wie folgt formulieren:

Der LP des k-ten Teilmodells habe die Elemente $(t_1^k, e_1^k, p_1^k) \ldots (t_j^k, e_j^k, p_j^k)$ seiner Ereignisfolge bereits ausgeführt. Wann kann er sicher sein, daß das vorderste Element seiner Ereignisliste $(t_{j+1}^k, e_{j+1}^k, p_{j+1}^k)$ ist?

Eine einfache Antwort auf diese Frage lautet: Dann, wenn er das Ereignis mit dem kleinsten Zeitstempel *aller* Ereignislisten *aller* LPs besitzt (bzw. das von π ausgewählte Ereignis, falls es mehrere solche gibt). Diese Lösung des Time-Advancement-Problems führt aber im allgemeinen dazu, daß zu jedem Zeitpunkt immer nur ein *einziges* Ereignis ausgeführt wird, so daß man statt paralleler Simulation eine über mehrere Rechnerknoten verteilte sequentielle Simulation erhält. Ein gewisses Maß an Parallelismus ist dann möglich, wenn π keine Rolle spielt, d. h. S_n für beliebige π dieselbe Ereignisfolge erzeugt. In diesem Fall können alle LPs, die ein Ereignis mit minimalem Zeitstempel besitzen, dieses ausführen. In [Sou92] findet man ein realistisches Beispiel (VLSI-Simulation mit einheitlicher Signallaufzeit), wo mit dieser Methode tatsächlich Geschwindigkeitsgewinne gegenüber sequentiellen Simulationen erzielt wurden. Die Methode setzt aber grundsätzlich voraus, daß in der

[1] Dieser Begriff stammt ursprünglich aus [Mis86] als Gegensatz zu dem dort verwendeten Begriff des physischen Prozesses, ist aber mittlerweile in der Literatur allgemein gebräuchlich und wird daher auch hier durchgängig verwendet.

Ereignisfolge von S_n immer wieder ausreichend viele Ereignisse mit identischen Zeitstempeln auftreten und diese auch noch zu unterschiedlichen Teilmodellen gehören. Im übrigen gibt es zwei grundsätzliche Lösungen des Time-Advancement-Problems:

Bei den *konservativen Synchronisationsverfahren* versucht jeder LP untere Schranken für die Zeitstempel der Ereignisse zu finden, die er noch von anderen LPs eingeplant bekommen kann. Liegt der Zeitstempel des vordersten Ereignisses seiner Ereignisliste unter dieser Schranke, so kann er das Ereignis ausführen, andernfalls muß er so lange warten, bis er eine neue (höhere) Schranke gefunden hat.

Bei den *optimistischen Synchronisationsverfahren* geht ein LP grundsätzlich davon aus, daß er das vorderste Ereignis seiner Ereignisliste ausführen darf. Bevor er dies tut, legt er jedoch eine Kopie der Datenstrukturen seines Teilmodells an. Stellt sich dann später heraus, daß er ein oder mehrere Ereignisse zu früh ausgeführt hat, so werden die Datenstrukturen auf den Stand nach der letzten korrekten Ausführung eines Ereignisses gebracht, irrtümlich für sich und andere LPs erzeugte Ereignisse annulliert und anschließend alle verfrüht ausgeführten Ereignisse erneut ausgeführt.

Beide Verfahren werden weiter unten noch ausführlich beschrieben. Auf eine Besonderheit des Time-Advancement-Problems im Zusammenhang mit gleichzeitigen Ereignissen soll jedoch hier schon näher eingegangen werden: Man stelle sich vor, ein LP hätte für ein gewisses t alle Ereignisse der Ereignisfolge seines Teilmodells mit Zeitstempeln $\leq t$ empfangen und diejenigen mit Zeitstempeln $< t$ bereits ausgeführt. Da die Ausführung eines Ereignisses nur von den Datenstrukturen des LPs und dem auszuführenden Ereignis abhängt, kann der LP auch alle Ereignisse mit Zeitstempel t ausführen, wobei er sich allerdings an die durch π vorgegebene Reihenfolge halten muß. Das Problem ist dabei jedoch, daß er diese Reihenfolge nicht ausschließlich aus den lokal bei ihm vorhandenen Ereignissen berechnen kann, da π sämtliche zum Simulationszeitpunkt t bei *allen* LPs vorhandenen Ereignisse berücksichtigt. Folglich kann ein LP nur dann beurteilen, ob die Ausführung eines Ereignisses korrekt ist, wenn er sämtliche Ereignisse mit Zeitstempel $\leq t$ der Ereignisfolge von S_n kennt. Somit müßten sich sämtliche LPs ständig über die im gesamten Simulationssystem vorhandenen Ereignisse informieren, was zu einem unvertretbar hohen Aufwand führen würde. Dies läßt sich umgehen, wenn man für π fordert, daß für jedes Teilmodell die Reihenfolge der Ereignisse mit identischen Zeitstempeln in seiner Ereignisfolge unabhängig von den Ereignissen anderer Teilmodelle ist. Präzise läßt sich das wie folgt formulieren:

Sei $k \in \mathbb{N}$ und $(t_i^k, e_i^k, p_i^k)_{a \leq i \leq b}$ eine endliche Teilfolge der Ereignisfolge des k-ten Teilmodells von S_n mit

$$t_i = t_{i+1} \; \forall \; a \leq i \leq b$$

sowie

$$t_b \neq t_{b+1}.$$

Weiterhin sei $M \subseteq (\bigcup_{i \neq k} E_i) \times P$ beliebig, aber endlich.

Ist

$$\pi(t_a, M \cup \{(e_a, p_a), \ldots, (e_b, p_b)\}) \in \{(e_a, p_a), \ldots, (e_b, p_b)\},$$

so ist

$$\pi(t_a, M \cup \{(e_a, p_a), \ldots, (e_b, p_b)\}) = (e_a, p_a).$$

Diese Forderung stellt in der Praxis im Regelfall keine Einschränkung dar, komplexe Beispiele für π, die dieses Kriterium erfüllen, findet man in [Meh91, CW91]. Ab sofort wird daher grundsätzlich vorausgesetzt, daß π diese Eigenschaft erfüllt.

Abschließend sei noch erwähnt, daß das hier vorgestellte Parallelisierungskonzept sich auf Parallelrechnern sowohl mit gemeinsamem Speicher als auch mit verteiltem Speicher realisieren läßt. Im wesentlichen besteht der Unterschied nur darin, daß das Einplanen und Löschen von Ereignissen zwischen zwei verschiedenen LPs bei Systemen mit verteiltem Speicher nicht direkt, sondern über spezielle Ereignis- bzw. Lösch-Nachrichten vonstatten geht. Auf Unterschiede bei den verschiedenen Simulationsverfahren wird ggf. hingewiesen, ansonsten sind auch sie von der Existenz gemeinsamen Speichers unabhängig.

3.2 Konservative Verfahren

Wie im vorhergehenden Abschnitt bereits erwähnt, versucht bei den konservativen Verfahren jeder LP, eine möglichst gute untere Schranke für die Zeitstempel der Ereignisse, die er noch von anderen LPs eingeplant bekommen kann, zu ermitteln. Eine solche Schranke läßt sich bereits aus den Ereignissen, die er eingeplant bekommen hat, ermitteln: Führt nämlich der LP i das Ereignis (t, e, p) aus und plant dabei dem LP j ein Ereignis ein (oder löscht eines seiner Ereignisse), so werden, da LP i in Zukunft nur noch Ereignisse mit Zeitstempeln $\geq t$ ausführen wird, LP j von LP i in Zukunft nur noch Ereignisse mit Zeitstempeln $\geq t$ eingeplant (bzw. gelöscht).

Diese Überlegungen führen zu dem in der Literatur unter dem Begriff *Linktime* bekannten Verfahren, das auf Chandy, Misra [Mis86] und Bryant [Bry79] zurückgeht. Es ist das Basisverfahren, auf dem im wesentlichen alle anderen konservativen Verfahren aufbauen. Bei diesem Verfahren betrachtet man ein Simulationssystem als einen Graphen, dessen Knoten die LPs sind und bei dem eine Kante vom LP des i-ten zum LP des j-ten Teilmodells von $\mathcal{S}_n$ geht, wenn $(t, e, p) \in \mathbb{R} \times E_i \times P$

und $(t', e', p') \in \mathbb{R} \times E_j \times P$ mit $(t', e', p') \in \tau(t, e, p)$ existieren, d. h. also, wenn die entsprechenden LPs sich prinzipiell Ereignisse einplanen können. Diese Kanten werden als Kanäle bezeichnet. Jeder LP führt nun für jeden Kanal , der in ihn hineinführt, eine Variable, die sogenannte *Kanaluhr (channel clock)*, die jedes Mal, wenn der LP ein Ereignis von dem LP am anderen Ende des Kanals eingeplant oder gelöscht bekommt, auf den Zeitstempel des erzeugenden Ereignisses gesetzt wird. Für einen LP mit den Kanaluhren $uhr_1, \ldots, uhr_k$ lautet die Antwort auf das Time-Advancement-Problem dann:

Alle Ereignisse mit Zeitstempeln echt kleiner als $\min\{uhr_1, \ldots, uhr_k\}$ dürfen ausgeführt werden[2].

Man beachte, daß durch die Forderung "... echt kleiner ..." garantiert ist, daß, falls ein LP mehrere Ereignisse mit gleichem Zeitstempel ausführen muß, er die korrekte Reihenfolge der Ausführung dieser Ereignisse mittels π bestimmen kann.

Ein generelles Problem mit dem Linktime-Verfahren ist, daß es bei Simulationssystemen, bei denen der Graph der LPs mit ihren Kanälen Zyklen enthält, sehr leicht zu Deadlocks kommen kann (existieren solche Zyklen nicht, so ist das Verfahren Deadlock-frei, vgl. [Su89]). Abb. 3.1 zeigt ein einfaches Beispiel: Hier enthält nur noch LP 2 zwei Ereignisse mit den Zeitstempeln 10 und 11, er dürfte somit beide Ereignisse nacheinander ausführen. Das Minimum der Kanaluhren seiner Eingangskanäle ist jedoch 9, daher kann er nur blockieren und darauf warten, durch Ankunft einer neuen Ereignisnachricht eine bessere Garantie zu bekommen. Da alle anderen Simulatoren ohnehin passiv sind, ist damit das Gesamtsystem verklemmt.

Chandy und Misra schlagen in [CM81] eine Lösung des Problems vor, die mittlerweile in der Literatur den Namen *Deadlock-Resolution* bekommen hat. Hierbei läßt man parallel zur eigentlichen Simulation einen parallelen Deadlock-Erkennungsalgorithmus laufen. Stellt dieser Algorithmus einen Deadlock fest, so bestimmt man (unter Berücksichtigung von π) das Ereignis, das den kleinsten Zeitstempel des gesamten Simulationssystems besitzt, setzt sämtliche Kanaluhren aller LPs auf dessen Zeitstempel (natürlich nur, falls sie nicht schon größer sind), und löst den Deadlock auf, indem man die Ausführung dieses Ereignisses gestattet. Was in der Literatur seltsamerweise nie erwähnt wird, ist die Tatsache, daß es dabei höchstens zur parallelen Ausführung gleichzeitiger Ereignisse kommen kann. Genauer gilt folgender Satz:

Satz 1 *Gerät ein Simulationssystem, das mit dem Deadlock-Resolution-Verfahren*

[2] Für Systeme mit verteiltem Speicher muß man hierbei allerdings zusätzlich fordern, daß sich beim Einplanen von Ereignissen via Nachrichtenverschickung zwischen zwei LPs die Ereignisnachrichten nicht überholen können.

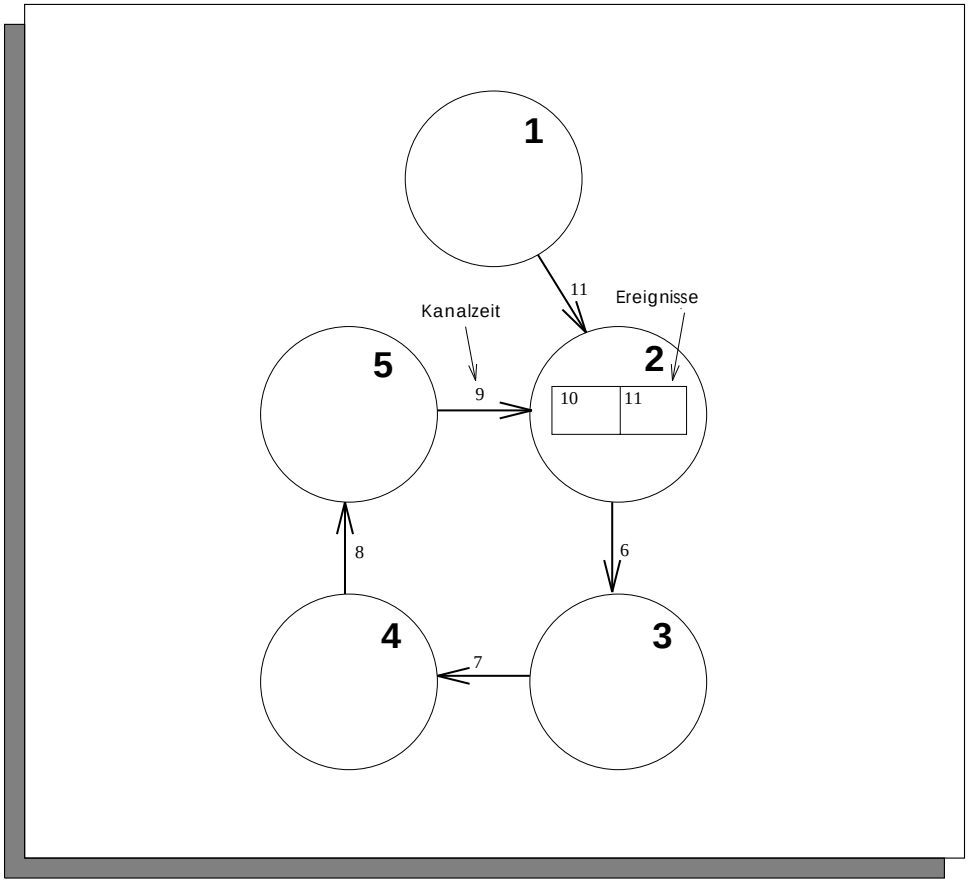


Abbildung 3.1: Deadlock bei konservativen Verfahren

arbeitet, einmal in einen Deadlock, so werden danach in den Phasen zwischen zwei Deadlocks nur noch Ereignisse mit identischen Zeitstempeln abgearbeitet.

Beweis: Man betrachte das Simulationssystem direkt nach dem Auflösen eines Deadlocks, wobei der kleinste Zeitstempel aller vorhandenen Ereignisse t sei. Dann gilt:

- Die vorhandenen Kanaluhren reichen nicht aus, um bei irgendeinem LP ein Ereignis mit einem Zeitstempel echt größer als t auszuführen.

  Begründung: Angenommen, ein LP besäße ein Ereignis (t', e, p) mit $t' > t$ und das Minimum seiner Kanaluhren wäre echt größer als t' und damit auch echt größer als t. Da während der Phase der Deadlock-Auflösung die Kanaluhren aller LPs maximal auf t hochgesetzt werden, muß das Minimum der Kanaluhren dieses LPs schon vor dem Deadlock echt größer als t' gewesen sein. Also

hätte das Ereignis auch schon vor dem Deadlock ausgeführt werden können. Widerspruch!

- Die Ausführung eines Ereignisses mit Zeitstempel t ändert keine Kanaluhr.

  Begründung: Wird bei der Ausführung eines Ereignisses mit Zeitstempel t einem anderen LP ein Ereignis eingeplant, so könnte dies lediglich die Kanaluhr des entsprechenden Kanals auf den Wert t hochsetzen, auf dem sie aber bereits seit dem Brechen des Deadlocks steht.

Aus diesen beiden Punkten ergibt sich die Behauptung. □

Somit führt bei Simulationssystemen, bei denen das Linktime-Verfahren nicht frei von Deadlocks ist, Deadlock-Resolution letztendlich zu dem im vorigen Abschnitt bereits geschilderten Verfahren. Nur wenn noch zusätzliche Informationen wie z. B. der weiter unten beschriebene Lookahead mit benutzt werden, macht es überhaupt erst von der Theorie her Sinn! Erwähnt sei dazu abschließend noch, daß das Verfahren ursprünglich in [CM81] für synchrone Nachrichtenkommunikation zwischen LPs vorgeschlagen wurde. In diesem Fall kommt noch eine weitere Sorte von Deadlocks, nämlich Kommunikationsdeadlocks, dazu, die mit dem geschilderten Verfahren natürlich auch aufgelöst werden. Allerdings stellt das Realisieren asynchroner Kommunikation auf den heutigen Parallelrechnern im allgemeinen kein Problem mehr dar.

Ein anderer Weg wird beim *Deadlock-Avoidance-Verfahren* beschritten, das ebenfalls von Chandy und Misra stammt [Mis86]. Bei diesem Verfahren geben sich die LPs zusätzliche Garantien in Form sogenannter Nullnachrichten (null messages). Nullnachrichten bestehen im wesentlichen nur aus einem Simulationszeitstempel (und enthalten kein Ereignis, daher der Name). Übergibt LP i an LP j eine Nullnachricht mit Zeitstempel t, so garantiert LP i, daß er LP j kein Ereignis mit einem Zeitstempel kleiner als t einplanen wird[3]. Das Deadlock-Avoidance-Verfahren, wie es in [Mis86] beschrieben wird, unterscheidet sich dann vom Linktime-Verfahren in zwei Punkten:

- Nach dem Ausführen eines Ereignisses übergibt der ausführende LP an die LPs, denen er kein Ereignis eingeplant hat und zu denen von ihm ein Kanal ausgeht, eine Nullnachricht mit dem Zeitstempel des ausgeführten Ereignisses.

[3] Beim "Übergeben" von Nullnachrichten auf Systemen mit verteiltem Speicher via Nachrichten-Verschicken muß man dabei wiederum Sorge tragen, daß sich Ereignis- und Nullnachrichten nicht gegenseitig überholen können.

- Erhält ein LP eine Nullnachricht, so setzt er die entsprechende Kanaluhr auf diesen Wert hoch und errechnet daraus eine neue Garantie (Minimum der Kanaluhren und des Zeitstempels des vordersten Ereignisses seiner Ereignisliste), die er über alle seine Ausgangskanäle weitergibt.

Dieses Verfahren allein würde jedoch noch keine Deadlockfreiheit garantieren. Es tut dies tatsächlich auch nur für spezielle Simulationssysteme, nämlich solche, bei denen die LPs die Möglichkeit haben, die Zukunft ein Stück weit vorauszusagen. Fujimoto hat dafür in [Fuj88] den Begriff des Lookahead geprägt. Mit dem hier eingeführten Formalismus läßt er sich leicht definieren:

Defininition 6 *Das k-te Teilmodell von S_n hat den Lookahead $\epsilon \in \mathbb{R}$, falls gilt*

$$t_{i+1}^k \geq t_i^k + \epsilon \quad \forall i \in \mathbb{N}$$

Das k-te Teilmodell hat also den Lookahead ϵ, falls alle Glieder seiner Ereignisfolge den zeitlichen "Mindestabstand" ϵ besitzen. Daher kann der zugehörige LP, falls er ein Ereignis mit dem Zeitstempel t ausführt, seinen Ereignissen und Nullnachrichten die Garantiezeit $t + \epsilon$ mitgeben. Weiterhin kann er beim Empfang einer Nullnachricht seine neu berechnete Garantie (also das Minimum der Kanaluhren und des Zeitstempels des vordersten Ereignisses seiner Ereignisliste) um ϵ erhöhen, bevor er sie in Form von Nullnachrichten über seine Ausgangskanäle weitergibt. In [Mis86] wird bewiesen, daß das Deadlock-Avoidance-Verfahren frei von Deadlocks ist, falls die LPs Garantien unter Berücksichtigung des Lookaheads geben und falls im Graph der LPs mit ihren Kanälen für jeden Zyklus ein LP existiert, dessen Teilmodell einen Lookahead > 0 besitzt.

Beispiele für Simulationsmodelle, bei denen das Deadlock-Avoidance-Verfahren anwendbar ist, sind Warteschlangennetze, bei denen eine untere Schranke für die Bedienzeiten ihrer Warteschlangen existieren (z. B. bei konstanter Bedienzeitverteilung). Hier ist der Lookahead gerade diese untere Schranke, da sie die minimale Differenz der Zeitstempel zweier Abgangsereignisse des zur Warteschlange gehörenden LPs darstellt. Bei Warteschlangennetzen, die Warteschlangen mit normal- oder exponentialverteilten Bedienzeiten besitzen, ist das Verfahren jedoch nicht anwendbar, da hier die Bedienzeiten beliebig kurz und daher Abgangsereignisse beliebig schnell aufeinander folgen können. Eine Lösung dieses Problems gibt Nicol in [Nic88] an: Dort wird die Tatsache ausgenutzt, daß Bedienzeiten auf Pseudozufallszahlengeneratoren beruhen und somit vorausberechenbar sind. Dies wird dazu benutzt, bei einem Abgangsereignis den frühest möglichen Zeitpunkt des darauffolgenden Abgangsereignis zu berechnen. Leider funktioniert diese Strategie nur für Warteschlangen mit FIFO-Bedienstrategie.

Abschließend sei noch erwähnt, daß es eine ganze Reihen von Varianten der bisher geschilderten Verfahren gibt. Beispiele sind etwa das Senden von Nullnachrichten nur vor dem Blockieren des sendenden LPs oder nur auf Anfrage des empfangenden LPs. Derartige Varianten sind sinnvoll, da im geschilderten Grundprinzip des Deadlock-Avoidance-Verfahrens unverhältnismäßig viele Nullnachrichten erzeugt werden, was zu Lawineneffekten und Kettenreaktionen durch Rückkoppelungen führen kann. Eine Übersicht darüber findet man in [Fuj90].

3.3 Optimistische Verfahren

Die Antwort der optimistischen Verfahren (die ihren Ursprung in den Arbeiten von Jefferson haben, vgl. insbesondere [Jef85]) auf das Time-Advancement-Problem ist zunächst einmal sehr einfach. Sie lautet: Falls die Ereignisliste nicht leer ist, führe ihr vorderstes Ereignis aus.

Dies kann natürlich dazu führen, daß ein Ereignis zu früh und damit auf Datenstrukturen mit falschen Werten ausgeführt wird. Bei den optimistischen Verfahren werden daher Vorkehrungen getroffen, die verfrühte Ausführung von Ereignissen rückgängig zu machen.

Abb. 3.2 zeigt den schematischen Aufbau eines LPs, der mit einem optimistischen Verfahren arbeitet: Hier enthält die Ereignisliste nicht nur die noch auszuführenden, sondern auch die bereits ausgeführten Ereignisse. Dies ist notwendig, da diese Ereignisse bei zu früher Ausführung eventuell später erneut (auf einem "korrigierten" Zustand) ausgeführt werden müssen. Weiterhin wird nach der Ausführung jedes Ereignisses eine Kopie der Werte der globalen Datenstrukturen des simulierten Teilmodells angelegt und mit dem Zeitstempel des zuletzt ausgeführten Ereignisses in einer nach diesem Zeitstempel sortierten Liste gespeichert. Schließlich werden in einer weiteren Liste Kopien sämtlicher neu erzeugter Ereignisse nach dem Zeitstempel des erzeugenden Ereignisses sortiert verwaltet.

Wird nun einem LP ein Ereignis eingeplant, das in der Ereignisfolge des entsprechenden Teilmodells vor dem zuletzt auf dem LP ausgeführten Ereignis kommt (ein sogenannter *Straggler*), so muß der LP in den Zustand zurückversetzt werden, in dem er nach dem letzten korrekt ausgeführten Ereignis (also dem letzten Ereignis, das in der Ereignisliste des LP vor dem Straggler kommt) gewesen wäre. Dieser Vorgang heißt *Rollback* und besteht aus folgenden Schritten (vgl. Abb. 3.3):

- Der Straggler wird in die Ereignisliste eingefügt. Alle bereits ausgeführten Er-

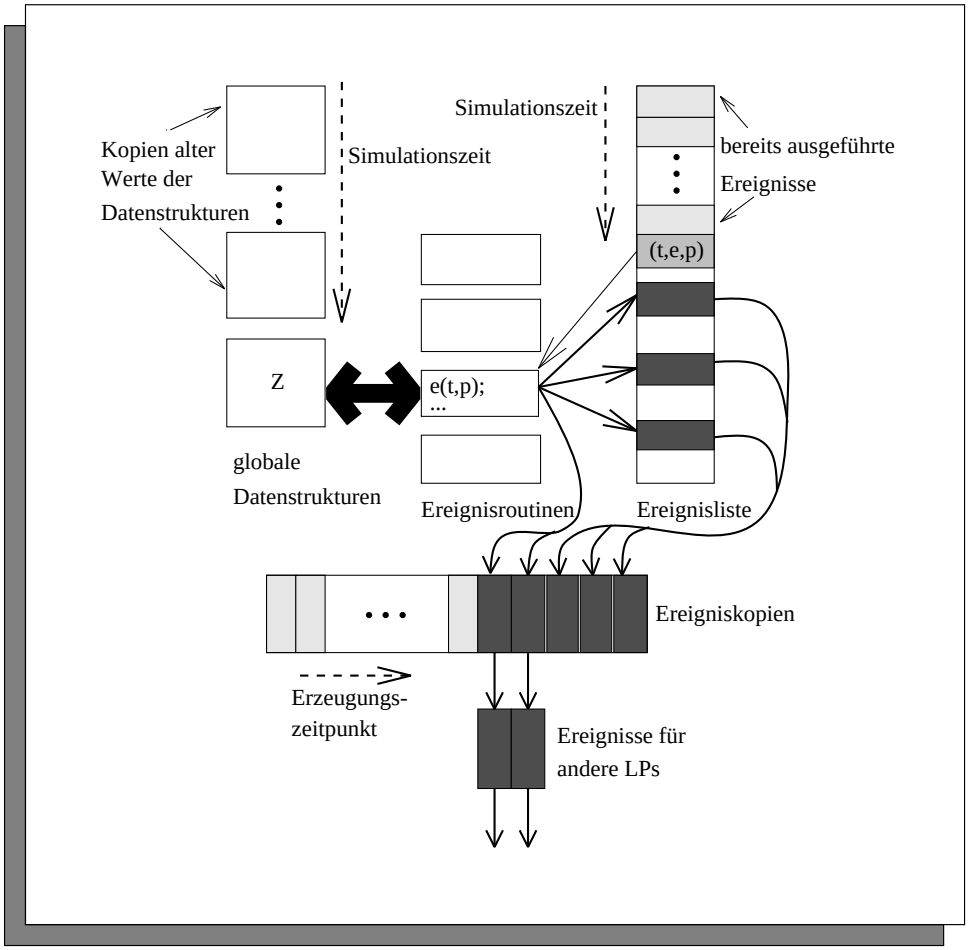


Abbildung 3.2: Optimistisch arbeitender LP

eignisse, die nach ihm in der Ereignisliste kommen, müssen erneut ausgeführt werden, da ihre Ausführung verfrüht war.

- Aus der Liste der Kopien der Datenstrukturen wird der Eintrag, der zu dem zuletzt korrekt ausgeführten Ereignis gehört, auf die aktuelle Version der Datenstrukturen kopiert.

- Durch die verfrühte Ausführung von Ereignissen können sowohl dem LP selber als auch anderen LPs irrtümlich neue Ereignisse eingeplant worden sein.

  Um dies rückgängig zu machen, wird für jeden Eintrag in der Liste der Ereigniskopien, der von einem Ereignis erzeugt wurde, dessen Zeitstempel größer als der des Stragglers ist, entweder das zugehörige Ereignis aus der Ereignisliste des LPs entfernt, oder aber es wird, falls es sich um die Kopie eines Ereignisses für einen anderen LP handelt, diesem eine sogenannte *Antinachricht*

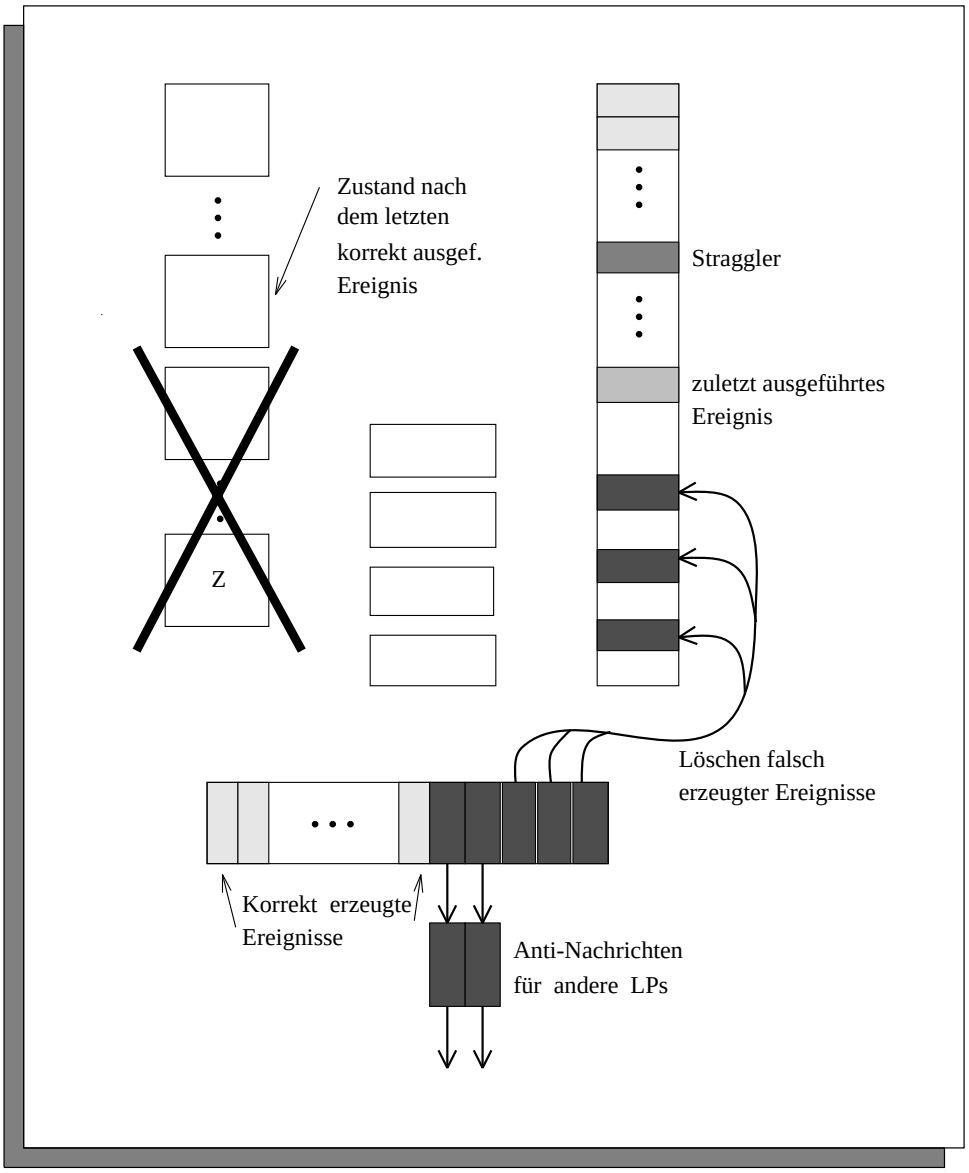


Abbildung 3.3: Rollback

eingeplant. Das Einplanen einer Antinachricht bewirkt bei dem betreffenden LP, daß das zugehörige Ereignis mit der Antinachricht zusammen gelöscht wird[4]. Wurde das gelöschte Ereignis bereits ausgeführt, so löst dies bei dem entsprechenden LP ebenfalls einen Rollback aus, der genauso durchgeführt wird, wie wenn das gelöschte Ereignis als Straggler eingetroffen wäre.

Der entsprechende Eintrag in der Liste der Ereigniskopien wird nach dem Erzeugen der Antinachricht gelöscht.

Man beachte, daß bei der Beschreibung des Rollbacks die Antwort auf die Frage, ob ein Ereignis zu früh ausgeführt worden ist, nur von der Position des Ereignisses in der Ereignisliste relativ zum Straggler abhängt. Somit ist auch der Fall berücksichtigt, daß außer dem Straggler auch noch andere Ereignisse mit demselben Zeitstempel vorhanden sein können, die in Abhängigkeit von π entweder korrekterweise oder aber zu früh ausgeführt wurden.

Ein anderes Problem stellen Ereignisse dar, die das Löschen anderer Ereignisse bewirken, da durch verfrühtes Ausführen von Ereignissen eventuell andere Ereignisse irrtümlich gelöscht werden können. In [LDF91] werden dazu zwei unterschiedliche Möglichkeiten vorgestellt:

Die Grundidee der ersten Lösung besteht darin, das entsprechende Ereignis erst zu kopieren und es dann durch Verschicken einer Antinachricht (beim Löschen über LP-Grenzen hinweg) oder direkt aus der Ereignisliste des LPs zu löschen. Die Kopien dieser Ereignisse werden in einer Liste sortiert nach dem Zeitstempel des Ereignisses, das die Löschung veranlaßt hat, gespeichert. Im Falle eines Rollbacks werden dann alle Ereignisse dieser Liste, die zu verfrüht ausgeführten Ereignissen gehören, erneut dem entsprechenden LP eingeplant und aus der Liste gelöscht. Letztendlich ist dies die zum Einplanen neuer Ereignisse analoge Vorgehensweise.

Bei der zweiten Lösung wird das gelöschte Ereignis in der Ereignisliste gelassen und dem entsprechenden LP ein "Löschereignis" mit gleichem Zeitstempel direkt vor diesem eingeplant. Die Ausführung des Löschereignisses bewirkt dann, daß in den Datenstrukturen des Simulationsmodells ein Eintrag vorgenommen wird, der bewirkt, daß bei der Ausführung des gelöschten Ereignisses sofort erkannt wird, daß es gelöscht wurde und daraufhin die Ausführung abgebrochen wird. Die Löschereignisse und die Einträge in der Datenstruktur werden dabei (insbesondere beim Rollback) wie Ereignisse und Daten des Simulationsmodells behandelt. Somit han-

[4] Bei Systemen mit verteiltem Speicher ist hier noch eine Besonderheit zu beachten, falls sich Ereignis- und Antinachricht überholen können: Trifft eine Antinachricht vor dem zugehörigen Ereignis auf einem LP ein, so wird sie dort zwischengespeichert und löscht das entsprechende Ereignis sofort bei dessen Eintreffen. Dabei kann niemals ein Rollback ausgelöst werden.

delt es sich hier eigentlich um eine Änderung des Simulationsmodells, bei dem das Löschen von Ereignissen durch das Einplanen von Löschereignissen ersetzt wird.

Ein anderes Problem ist die Frage, ob bei einem Simulationssystem, das nach einem optimistischen Verfahren arbeitet, immer gewährleistet ist, daß die Simulation nicht auf der Stelle tritt, d. h. daß alle LPs alle Ereignisse ihrer Ereignisfolge früher oder später ausführen. Um diese Frage präzise beantworten zu können, muß zunächst noch ein Begriff neu eingeführt werden, der auch für andere Aspekte der optimistischen Verfahren von zentraler Bedeutung ist, nämlich der Begriff der globalen virtuellen Zeit (global virtual time, daher im folgenden wie allgemein üblich GVT abgekürzt). Dazu stelle man sich vor, jeder LP besitze eine Variable, die vor der Ausführung eines Ereignisses auf dessen Zeitstempel und nach der Ausführung des letzten in der Ereignisliste vorhandenen Ereignisses auf unendlich gesetzt werde. Die GVT ist dann genau das Minimum dieser Variablen[5]. Für sie gilt jederzeit:

- Die Zeitstempel von Stragglern und Antinachrichten sind immer mindestens so groß wie die GVT. Ausgeführte Ereignisse mit Zeitstempeln kleiner als die augenblickliche GVT müssen daher nie wieder erneut ausgeführt werden.
- Der Wert der GVT kann immer nur steigen, niemals fallen.
- Unter den Voraussetzungen, daß in der Ereignisfolge des Simulationsmodells jeder Zeitstempel nur endlich oft vorkommt und jeder LP solange Ereignisse ausführt, wie seine Ereignisliste nicht leer ist, behält die GVT einen endlichen Wert nur endlich lang.

Diese Eigenschaften wurden informell bereits in [Jef85] gezeigt. In [LW93] werden sie formal für ein abstraktes Berechnungsmodell bewiesen, das als Spezialfall das hier vorgestellte optimistische Verfahren enthält. Dies geschieht dort unter Voraussetzung gewisser Fairneß-Bedingungen, die hier aber alle erfüllt sind. Somit kann ein Ereignis als endgültig ausgeführt betrachtet werden, wenn die GVT des Simulationssystems echt größer als der Zeitstempel des Ereignisses ist. Weiterhin ist bei Simulationsmodellen mit endlicher Ereignisfolge die Simulation genau dann beendet, wenn die GVT den Wert unendlich erreicht. Somit läßt sich die weiter oben gestellte Frage präziser und schärfer wie folgt formulieren:

Gibt es bei einem nach einem optimistischen Verfahren arbeitenden Simulationssystem immer für jedes Ereignis der Ereignisfolge des Simulationsmodells einen

[5] Bei Systemen mit verteiltem Speicher ist die GVT das Minimum der Variablen *und* der Zeitstempel der Ereignisse und Antinachrichten, die "unterwegs" (d.h. abgesendet aber noch nicht angekommen) sind.

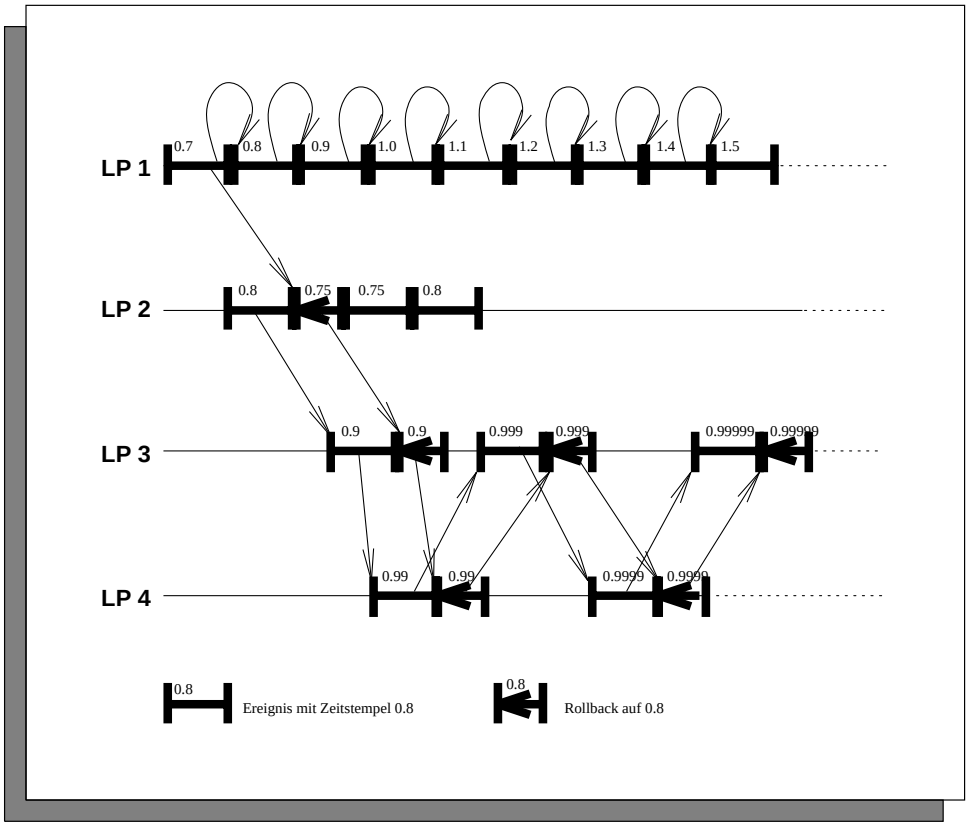


Abbildung 3.4: Der Dog-chasing-its-tail-Effekt

(Real-)Zeitpunkt, ab dem die GVT des Simulationssystems dessen Zeitstempel überschreitet? Erreicht dabei für Simulationsmodelle mit endlichen Ereignisfolgen die GVT immer den Wert unendlich in endlicher (Real-)Zeit?

Die Antwort auf beide Fragen lautet im allgemeinen leider nein! Der Grund dafür ist in der Literatur als "Dog-chasing-its-tail-Effekt" [Fuj90] bekannt und wird in [LW93] als Wirbel (vortex) bezeichnet. Er kann in zwei Situationen auftreten:

1.) Ein verfrüht ausgeführtes Ereignis erzeugt einen Schwarm von Ereignissen auf anderen LPs, der sich so schnell ausbreitet, daß die Antinachrichten es nie schaffen, alle Ereignisse zu annullieren und bei dem die Zeitstempel seiner Ereignisse einen endlichen Häufungspunkt bilden. Die GVT wächst dann zwar monoton ihrem Häufungspunkt entgegen, kommt aber niemals über ihn hinaus. Abb. 3.4 zeigt ein solches Beispiel. LP 2 führt dort das Ereignis mit Zeitstempel 0.8 zu früh aus und plant daher LP 3 irrtümlich ein Ereignis mit Zeitstempel 0.9 ein, was dazu führt, daß sich zwischen LP 3 und LP 4 ein Schwarm von Ereignissen hin und her bewegt, bei dem die zugehörigen

Antinachrichten immer erst dann eintreffen, wenn das entsprechende Ereignis bereits ausgeführt ist. Die GVT nimmt dann die Werte 0.9, 0.99, 0.999, usw. an und kommt über den Wert 1.0 nicht hinaus.

2.) Ist die Ereignisfolge des Simulationsmodells endlich, so kann es passieren, daß ein ähnlicher Schwarm von Ereignissen und hinterherlaufenden Antinachrichten entsteht, deren Zeitstempel größer sind als der des letzten Elements der Ereignisfolge. Die GVT überschreitet dann zwar ebenfalls diesen Wert und alle Ereignisse der Ereignisfolge sind damit endgültig ausgeführt, jedoch erreicht die GVT nie den Wert unendlich, der Anwender kann also nie sicher sein, daß der Simulationslauf beendet ist.

Für diese Probleme gibt es zumindest zur Zeit keine zufriedenstellende Lösung. (Die in [LW93] angegebene funktioniert nur für einen sehr eingeschränkten Spezialfall.) In praktischen Realisierungen kann man die geschilderten Probleme jedoch im Regelfall umgehen, indem man dafür sorgt, daß Antinachrichten ihr Ziel schneller erreichen als Ereignisse (z. B. durch Prioritäten auf Nachrichtenpuffern). Weiterhin kann zumindest der unter 1.) geschilderte Effekt nicht auftreten, wenn zwischen den Zeitstempeln des erzeugenden und des erzeugten Ereignisses immer ein gewisser Mindestabstand vorhanden ist, d. h. wenn gilt

$$\exists d > 0 \;\; \forall i \;\; \forall (t, e, p) \in \mathbb{R} \times E_i \times P \;\; \forall (t', e', p') \in \tau(t, e, p) : \qquad t' - t \geq d$$

Dies gilt z. B. für Simulationsmodelle, bei denen nur ganzzahlige Zeitstempel auftreten.

Über die rein theoretischen Aspekte hinaus ist die GVT auch eine wichtige Größe für die Implementierung gewisser Teilaufgaben des Simulationssystems. So gibt es z. B. in konkreten Anwendungen irreversible Aktionen (wie etwa das Beschreiben von Dateien, Bildschirmausgaben, das Starten einer Rakete etc.). Die Ausführung dieser Aktionen muß dann solange verzögert werden, bis sichergestellt ist, daß das veranlassende Ereignis nicht erneut ausgeführt oder durch eine Antinachricht gelöscht wird, also solange, bis die GVT den Zeitstempel des veranlassenden Ereignisses überschreitet.

Eine weitere wichtige Anwendung stellt die Speicherverwaltung des Simulationssystems dar. Da kein Ereignis, dessen Zeitstempel kleiner als die aktuelle GVT ist, jemals wieder erneut ausgeführt wird, sind diese Ereignisse sowie die zugehörigen Kopien der Datenstrukturen und der erzeugten Ereignisse überflüssig und können gelöscht werden. Lediglich die Kopie der Datenstrukturen nach Ausführung des letzten Ereignisses, dessen Zeitstempel kleiner als die GVT war, muß für den Fall erhalten werden, daß das erste der verbleibenden Ereignisse zu früh ausgeführt wurde.

Dieser in regelmäßigen Abständen stattfindende Vorgang wird in der Literatur als *Fossil-Collection* bezeichnet.

Sowohl für das Ausführen irreversibler Operationen als auch für die Fossil-Collection wird daher in regelmäßigen Abständen der Wert der GVT benötigt. Realisiert wird dies normalerweise, indem ein ausgezeichneter Knoten des Parallelrechners in regelmäßigen Abständen einen parallelen Algorithmus startet, der eine untere Schranke der aktuellen GVT bestimmt. Das Ergebnis dieses Algorithmus wird dann allen anderen Knoten des Rechners mitgeteilt, worauf diese die Fossil-Collection und ggf. gepufferte irreversible Operationen durchführen. Parallele Algorithmen, die eine untere Schranke für die GVT berechnen, gibt es mittlerweile reichlich (z. B. [Bel90, Sam85, LL90, Mat93], zumal sie sich auch aus anderen parallelen Algorithmen wie etwa den Terminierungsalgorithmen (vgl. [MMST91]) ableiten lassen. Da diese Algorithmen im allgemeinen nur geringen Einfluß auf das Laufzeitverhalten der Simulation haben (vgl. hierzu z. B. die Ausführungen in [Bel90]) und eine Darstellung der verschiedenen Algorithmen recht umfangreich würde, wird darauf hier verzichtet. In Kapitel 8 wird jedoch der im Rahmen dieser Arbeit verwendete Algorithmus beschrieben.

Bei der praktischen Realisierung optimistischer Simulationsverfahren gibt es im übrigen noch zwei Probleme, die immer wieder auftauchen:

Einmal sind die in der Praxis verwendeten Funktionen τ_i, φ_i und ρ_i vielfach partielle Funktionen, so daß es, falls eine Ereignisroutine mit den falschen Parametern aufgerufen wird, zu einem Fehler kommen kann. Genau dies kann aber bei der Ausführung von Ereignissen, die von zu früh ausgeführten Ereignissen irrtümlich erzeugt wurden, passieren. In diesem Fall muß der entsprechende LP nicht etwa wie bei einem echten Fehler im Simulationsmodell mit einer Fehlermeldung terminieren, sondern auf die zu dem fehlerhaften Ereignis gehörende Antinachricht warten (die, falls das Simulationsmodell keinen Fehler enthält, früher oder später eintreffen muß).

Ein anderes Problem besteht darin, daß durch das Speichern der Kopien der Datenstrukturen ein erheblicher Verbrauch an Hauptspeicher entsteht. Somit besteht (insbesondere da auf den im Augenblick verfügbaren Parallelrechnern meist kein virtueller Speicher verfügbar ist) die Gefahr, daß einem LP auf Grund von Speichermangel Ereignisse oder Antinachrichten von einem anderen LP nicht mehr eingeplant werden können. Blockiert ein LP in dieser Situation, so kann es z. B. passieren, daß ihn Antinachrichten, die wieder Speicher freigeben und damit den Fortgang der Simulation ermöglichen würden, nicht mehr erreichen, und das gesamte Simulationssystems in einen Deadlock gerät. Für dieses Problem existieren mehrere Lösungen [Gaf88, Jef90, Lin92a], die aber alle auf derselben Idee beruhen: Trifft ein Ereignis oder aber eine Antinachricht ein, für die selbst nach Durchführung der Fossil-

Collection kein Speicherplatz mehr vorhanden ist, so sendet entweder der Empfänger sie (oder ein anderes Ereignis bzw. eine andere Antinachricht) an den Sender zurück, was bei diesem einen Rollback auf den Erzeugungszeitpunkt der Nachricht bewirkt, oder aber er führt bei sich einen Rollback auf einen Zeitpunkt oberhalb der GVT durch, wodurch wieder Speicherplatz freigegeben wird. Ein Vergleich der verschiedenen Lösungen, die sich im wesentlichen darin unterscheiden, was zurückgeschickt bzw. welcher Zeitpunkt für den Rollback gewählt wird, findet sich in [LP91].

Bei den optimistischen Verfahren gibt es wiederum eine Fülle von Varianten, auf die aus Platzgründen hier nicht näher eingegangen werden soll. (Eine gute Übersicht ist wiederum [Fuj90] oder auch [Rös93]). Es werden hier jedoch abschließend noch zwei Varianten geschildert, die so fundamental sind, daß sie schon fast zum "Standard-Verfahren" dazugehören.

Die erste Variante betrifft das Speichern von Kopien der Datenstrukturen [JS82]. Um den durch sie verbrauchten Speicher zu reduzieren, kann man nur nach der Ausführung jedes n-ten Ereignisses eine Kopie anlegen, wodurch lediglich $1/n$ des ursprünglichen Speicherplatzbedarfs für Kopien benötigt wird. Andererseits müssen dann unter Umständen nach einem Rollback die Datenstrukturen nach der letzten korrekten Ausführung eines Ereignisses rekonstruiert werden, indem ausgehend von den letzten korrekten, gespeicherten Datenstrukturen die Ereignisse, zu denen keine Kopie gespeichert wurde, erneut ausgeführt werden. Man nennt dies die *Coasting-Forward-Phase*. Insgesamt erkauft man sich damit also Speichergewinn durch Rechenzeitverlust.

Die zweite Variante ist unter dem Begriff Lazy-Cancellation bekannt und geht auf Gafni [Gaf88] zurück. Die Idee ist, im Falle eines Rollbacks nicht sofort alle Antinachrichten, die zu verfrüht ausgeführten Ereignissen gehören, loszusenden, sondern zunächst die entsprechenden Ereigniskopien nur zu markieren. Anschließend wird jedes nach dem Rollback neu erzeugte Ereignis mit den vorhandenen markierten Ereigniskopien verglichen. Falls es eine identische, markierte Ereigniskopie gibt, wird lediglich bei dieser die Markierung entfernt, andernfalls wird das Ereignis dem entsprechenden LP eingeplant. Hat das vorderste nicht ausgeführte Ereignis der Ereignisliste einen Zeitstempel, der größer ist als der entsprechende Zeitstempel vor Ausführung des Rollbacks, so wird die Liste der Ereigniskopien nach markierten Ereignissen durchsucht und für diese Kopien entsprechende Antinachrichten erzeugt. Diese Variante hat Vorteile, wenn die erneute Ausführung von Ereignissen nach einem Rollback häufig dieselben Ereignisse erneut produziert, da auf diese Weise unnötige Rollbacks bei anderen LPs vermieden werden. Ein Beispiel dafür ist ein LP, der für die Simulation mehrerer verschmolzener Teilmodelle zuständig ist, da dort ein Straggler zunächst nur die Datenstrukturen seines Teilmodells verändert und somit die Ausführung von Ereignissen für andere Teilmodelle überhaupt nicht

beeinflußt.

3.4 Verfahren zur Parallelitätsanalyse

In diesem Abschnitt werden Verfahren vorgestellt, die dazu dienen, eine untere Schranke für die Laufzeit der parallelen Simulation eines vorgegebenen, verteilten ereignisgesteuerten Simulationsmodells zu bestimmen. Vergleicht man diesen Wert mit der Laufzeit einer sequentiellen Simulation des Simulationsmodells, bei dem alle Teilmodelle zu einem verschmolzen sind, so erhält man einen Anhaltspunkt, ob sich die parallele Simulation dieses Modells überhaupt "lohnt". Dies diente im Rahmen dieser Arbeit dazu, Vergleiche zwischen unterschiedlichen Simulationsverfahren nur mit "sinnvollen" Modellen durchzuführen, d. h. mit solchen Modellen, die nicht schon von ihrer Struktur her für parallele Simulation ungeeignet sind. Man kann dies in der Praxis aber auch dazu benutzen, vorhandene Simulationsmodelle auf ihre "Parallelitätstauglichkeit" zu prüfen, bevor für sie mit viel Aufwand ein paralleler Simulator realisiert wird.

Wie später noch an praktischen Beispielen gezeigt wird, können die Ausführungszeiten für verschiedene Ereignisse sehr unterschiedlich sein. Daher macht es wenig Sinn, Verfahren zur Parallelitätsanalyse zu konstruieren, die ohne diese Realzeitinformationen auskommen. Sie messen zu können setzt natürlich die Existenz mindestens eines Prozessors des später verwendeten Parallelrechners mit derselben Programmierumgebung (Compiler, Laufzeitsystem etc.) voraus. In der Praxis stellt diese Voraussetzung vielfach kein Hinderniss dar, insbesondere da, wie weiter unten noch näher erläutert wird, es unter Umständen auch genügt, lediglich das Verhältnis der verschiedenen Ereignisausführungszeiten zueinander zu kennen.

3.4.1 Analyse der Lastbalancierung und Kritische-Pfad-Analyse

Die Idee dieser Verfahren beruht darauf, lediglich auf Grund der Ausführungszeiten der Ereignisse sowie eventuell unter Berücksichtigung kausaler Abhängigkeiten zu berechnen, wie lange ein paralleler Simulationslauf dauern würde. Es wird dabei davon ausgegangen, daß jedes Teilmodell von einem LP auf einem eigenen Prozessor simuliert wird. Außerdem werden folgende, idealisierende Annahmen gemacht:

- Jedes Ereignis wird ausgeführt, sobald es für den entsprechenden LP verfügbar ist und der LP nicht vorher noch andere Ereignisse ausführen muß. (Es wird also beispielsweise vom Warten auf Garantien wie bei konservativen Synchronisationsverfahren abstrahiert)
- Die Ausführung eines Ereignisses dauert auf einem parallelen Simulator genauso lange wie auf einem sequentiellen Simulator (was z. B. den Aufwand für das Kopieren von Datenstrukturen bei den optimistischen Synchronisationsverfahren ignoriert).
- Das Einplanen von Ereignissen kostet den einplanenden LP keine Rechenzeit, der LP, dem das Ereignis eingeplant wird, erhält dieses in Nullzeit. (Hier werden z. B. Startzeiten und Nachrichten-Laufzeiten für das Versenden von Nachrichten bei Systemen mit verteiltem Speicher nicht berücksichtigt.)

Unter diesen Annahmen lassen sich zwei untere Schranken für die Laufzeit der parallelen Simulation des Simulationsmodells berechnen (Abb. 3.5):

Bei der *Analyse der Lastbalancierung* geht man (wiederum stark idealisierend) davon aus, daß jeder LP bereits bei Start der Simulation sämtliche auszuführende Ereignisse kennt und sie ohne Pause hintereinander ausführt. Die Laufzeit dieses idealisierten parallelen Simulationslaufs erhält man dann, indem man für jeden LP die Ausführungszeiten seiner Ereignisse aufsummiert und das Maximum dieser Summen bestimmt (t_{bal} in Abb. 3.5). Dieser Wert ist daher leicht mit Hilfe eines entsprechend instrumentierten sequentiellen (oder parallelen) Simulators zu bestimmen. Dividiert man diesen Wert durch die Summe der Ausführungszeiten aller Ereignisse (also die Laufzeit eines sequentiellen Simulationslaufs des Modells, t_{seq} in Abb. 3.5), so erhält man einen Wert, der eine obere Schranke für die mögliche Beschleunigung durch parallele Simulation gegenüber sequentieller Simulation darstellt. Wie nah dieser Wert der Anzahl der vorhandenen Prozessoren kommt sagt dann etwas über die Qualität der Lastbalancierung, also die Gleichmäßigkeit der Verteilung der Ausführungszeiten von Ereignissen auf die LPs, aus.

Bei der *Kritischen-Pfad-Analyse* wird im Gegensatz zu dem gerade geschilderten Verfahren noch zusätzlich berücksichtigt, daß (mit Ausnahme der beim Start der Simulation vorhandenen Ereignisse) ein Ereignis erst ausgeführt werden kann, nachdem es bei der Ausführung eines anderen Ereignisses erzeugt wurde. Da lediglich eine obere Schranke für die erzielbare Beschleunigung gesucht wird, wählt man dabei den Erzeugungszeitpunkt so früh wie möglich, nämlich am Anfang der Ausführung des erzeugenden Ereignisses (Abb. 3.5)[6]. Die Laufzeit dieses idealisierten Simula-

[6] Dies kann dann natürlich dazu führen, daß das erzeugende und das erzeugte Ereignis eine zeitlang parallel ausgeführt werden, was bei langen Ereignisausführungszeiten auch in der Praxis denk-

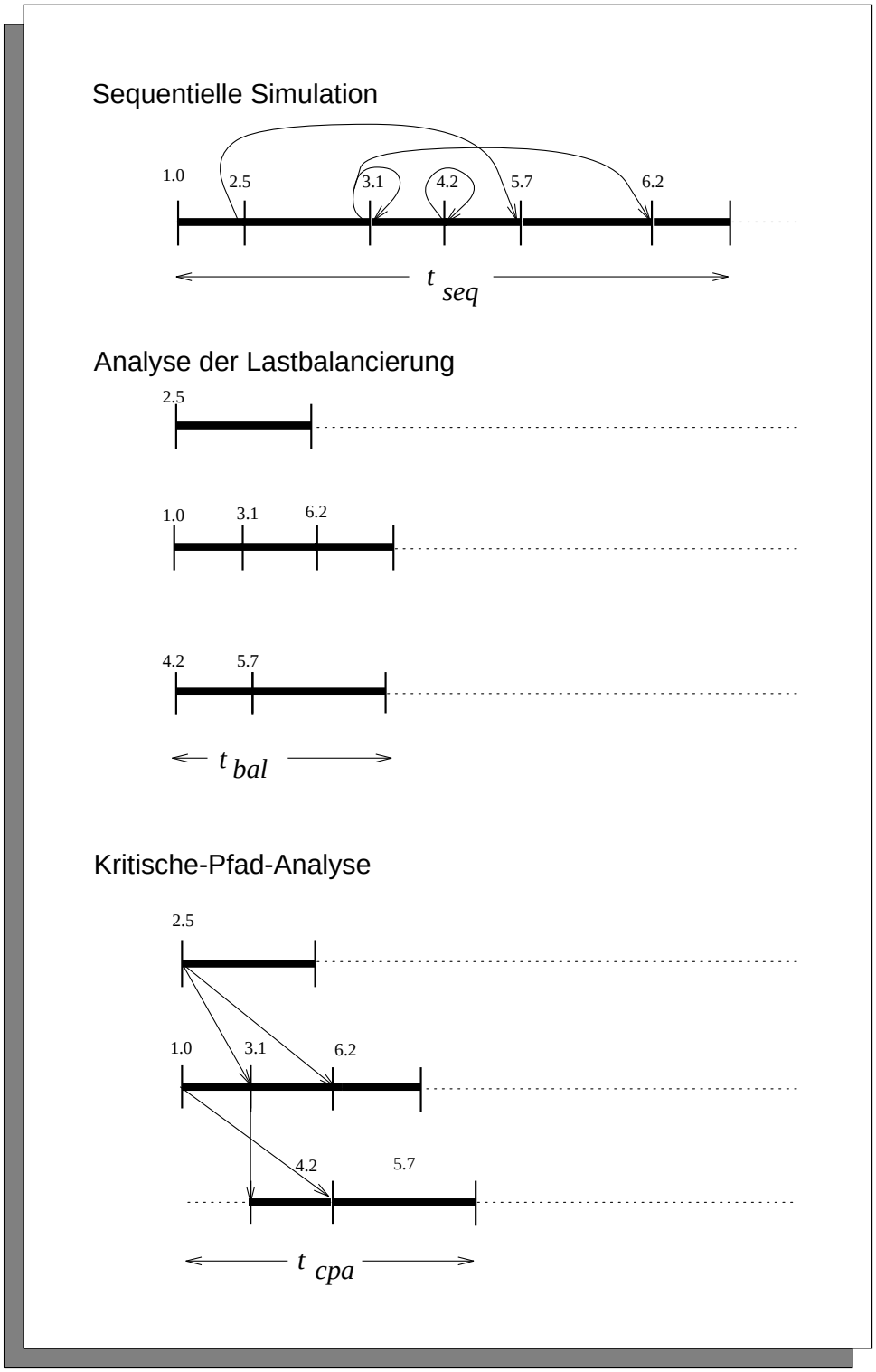


Abbildung 3.5: Analyse der Lastbalancierung und Kritische-Pfad-Analyse

tionslaufs ist dann gerade die Differenz zwischen dem Start der Ausführung des ersten und dem Ende der Ausführung des letzten Ereignisses durch einem LP (t_{cpa} in Abb. 3.5).

Stellt man sich einen Graphen vor, bei dem die Knoten die ausgeführten Ereignisse sind und bei dem eine Kante von Knoten a nach Knoten b geht, falls entweder a und b zu demselben Teilmodell gehören und b in dessen Ereignisfolge direkt vor a kommt oder aber falls b bei der Ausführung von a erzeugt wird und gewichtet man die Knoten mit den Ausführungszeiten der Ereignisse und die Kanten mit 0, so entsteht ein azyklischer Graph, bei dem die Länge des längsten Weges gerade t_{cpa} ist. Eine Möglichkeit, t_{cpa} zu berechnen, wäre daher, diesen Graph während eines sequentiellen Simulationslaufs aufzubauen und anschließend einen Algorithmus zur Bestimmung des kritischen Pfades darauf ablaufen zu lassen (z. B. mit einer Variante des in [Eve79] beschriebenen Algorithmus für PERT-Digraphen). Diese Methode wurde auch ursprünglich in [BJ85, SCS89] vorgeschlagen. Sie hat jedoch den Nachteil, daß dazu während eines sequentiellen oder parallelen Simulationslaufs der gesamte Graph zwischengespeichert werden muß, was bei längeren Läufen sehr platzaufwendig sein kann.

In [RW89] wird stattdessen eine erheblich einfachere Methode angegeben, mit der ein sequentieller Simulator ohne großen Aufwand instrumentiert werden kann. Für jedes Teilmodell m wird dort eine Variable $last_m$ geführt. Diese beschreibt den letzten bisher bekannten (hypothetischen) Realzeitpunkt, an dem der zugehörige LP mit der Ausführung eines Ereignisses begonnen oder aufgehört hat. Wird bei der Ausführung eines Ereignisses des Teilmodells m ein neues Ereignis erzeugt, so erhält es als Zusatzinformation den Wert von $last_m$ beim Start der Ausführung des erzeugenden Ereignisses. Vor der Ausführung eines Ereignisses wird dann für den entsprechenden LP $last_m$ auf das Maximum seines bisherigen und des mit dem Ereignis gespeicherten Wertes gesetzt (vgl. Abb. 3.6). Nach der Ausführung eines Ereignisses wird $last_m$ um die Ausführungsdauer des Ereignisses erhöht. Der gesucht Wert ist $\max_m\{last_m\}$ am Ende des Simulationslaufs. Ein Beweis für die Korrektheit dieses Algorithmus findet man in [Lin92b].

Dividiert man den so ermittelten Wert durch die Summe der Ausführungszeiten der Ereignisse t_{seq}, so erhält man eine schärfere obere Schranke für die durch parallele Simulation erzielbare Beschleunigung als bei der oben beschriebenen Analyse der Lastbalancierung. Durch den Vergleich dieser Schranke mit der aus der Analyse der Lastbalancierung gewonnenen hat man außerdem noch einen Anhaltspunkt, wie

bar ist. Letztendlich ist dieses Phänomen eine Folge der Tatsache, daß bei den hier durchgeführten Untersuchungen im Gegensatz zu dem beispielsweise in [Mat89] verwendeten Kausalitätsbegriff die Ausführung von Ereignissen nicht atomar ist, sondern eine zeitliche Länge besitzt.

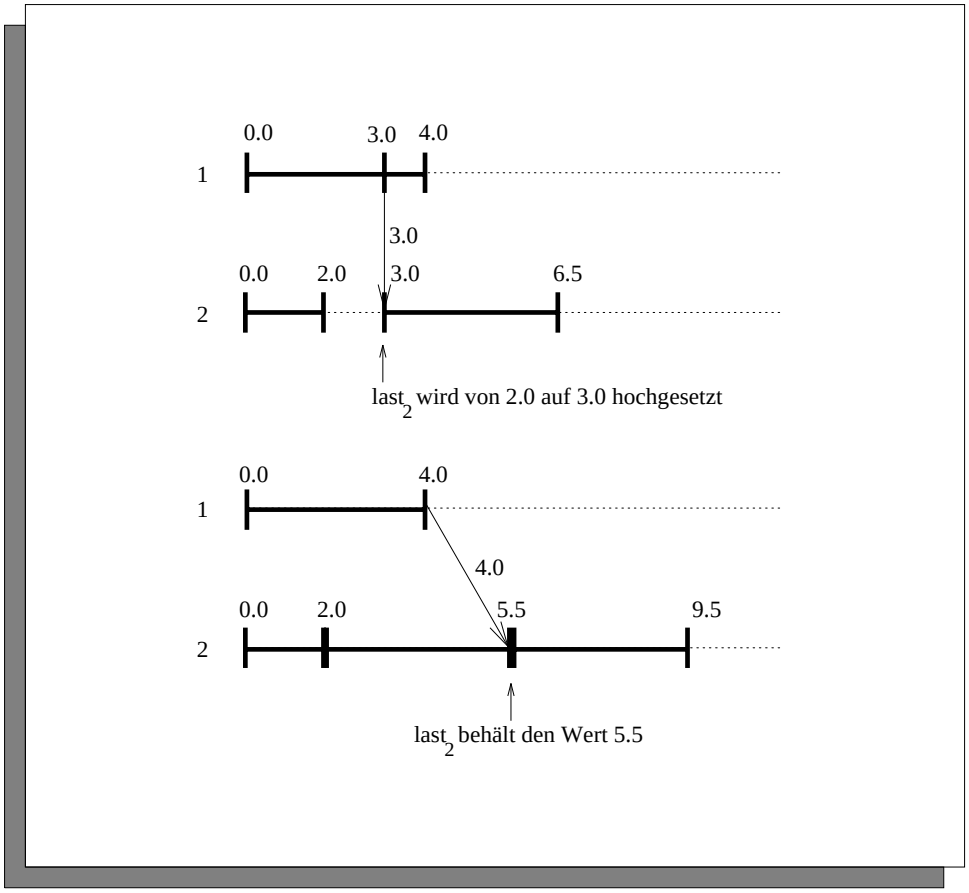


Abbildung 3.6: Vereinfachte Kritische-Pfad-Analyse

sehr kausale Abhängigkeiten bei der parallelen Simulation des spezifischen Modells eine Rolle spielen.

Abschließend sei noch angemerkt, daß die gerade beschriebenen Analysen noch eine interessante Stabilitätseigenschaft besitzen: Multipliziert man die Ausführungszeiten aller Ereignisse mit einem konstanten Faktor, so kürzt sich dieser in den Quotienten $\frac{t_{seq}}{t_{bal}}$ bzw. $\frac{t_{seq}}{t_{cpa}}$, die die Beschleunigungsschranken angeben, wieder heraus. Damit ist es im Prinzip möglich, diese Analysen auf einem System durchzuführen, bei dem sich die Ausführungszeiten der Ereignisse von denen des später verwendeten Parallelrechners um einen konstanten Faktor unterscheiden (beispielsweise bei einer Prozessoremulation).

3.4.2 Parallele Simulation mit Orakeldateien

Dieses Verfahren stammt aus [SF87] und beruht auf einem gänzlich anderen Ansatz als den bisher geschilderten. Es handelt sich dabei im Grunde um ein weiteres paralleles Simulationsverfahren. Bei ihm wird zunächst ein (paralleler oder sequentieller) Simulationslauf durchgeführt, bei dem alle ausgeführten Ereignisse in einer Datei (der Orakeldatei) protokolliert werden. Anschließend wird dasselbe Modell noch einmal parallel simuliert, wobei jeder LP anhand der Orakeldatei ermittelt, ob das vorderste Ereignis seiner Ereignisliste das nächste von ihm auszuführende Ereignis ist. Falls dies zutrifft, führt er es aus, falls nicht, wartet er auf das Eintreffen weiterer Ereignisse. Die Laufzeit des zweiten Simulationslaufs wird dann als untere Schranke für die Laufzeit realistischer (d. h. mit praktisch anwendbaren Simulationsverfahren durchgeführter) paralleler Simulationsläufe des Modells genommen. In ihr ist der generell zusätzlich zur Ereignisausführung anfallende Aufwand bei paralleler Simulation berücksichtigt, lediglich die von speziellen Verfahren zur Lösung des im Abschnitt 3.1 erläuterten Time-Advancement-Problems abhängenden Aspekte (z. B. Warten auf Garantien, Kopieren von Datenstrukturen etc.) werden nicht berücksichtigt.

Kritisch sei dazu jedoch angemerkt, daß dieses Verfahren zwei praktische Probleme aufwirft: Einmal muß man, um es durchzuführen, einen parallelen Simulator realisieren, d. h. nur mit der Instrumentierung eines existierenden sequentiellen Simulators ist das Verfahren nicht durchführbar. Zum zweiten entstehen bei diesem Verfahren große Mengen von Dateizugriffen, die effizient implementiert werden müssen, um den parallelen Simulationslauf nicht zu stark zu verfälschen. Hierauf wird noch in Kapitel 5 näher eingegangen.

Zusammen mit den im vorigen Abschnitt erläuterten Methoden hat man damit eine Reihe immer genauer werdender Schranken für die Laufzeit einer parallelen Simulation: Seien die für ein beliebiges Simulationsmodell mit der Lastbalancierungs-, Kritischen-Pfad- und Orakeldatei-Analyse gemessenen Laufzeiten mit t_{bal}, t_{cpa}, t_{oracle} bezeichnet, so gilt für die Laufzeit t_{par} einer beliebigen parallelen Simulation desselben Modells:

$$t_{bal} \leq t_{cpa} \leq t_{oracle} \leq t_{par}$$

Mit der Genauigkeit dieser Werte steigt jedoch auch der Aufwand für ihre Bestimmung und damit auch die Zahl der Möglichkeiten, die Messungen durch Ungenauigkeiten zu verfälschen. Dies kann in der Praxis in Ausnahmefällen sogar dazu führen, daß die obige Ungleichungskette nicht mehr gilt, wie sich in Kapitel 6 noch zeigen wird.

3.5 Beschleunigungsmessungen in der Praxis

Abschließend soll in diesem Abschnitt noch auf die Frage eingegangen werden, welche Beschleunigungen ereignisgesteuerter Simulationen durch parallele Simulation bereits anderweitig erreicht wurden. Neben generellen Problemen bei der Durchführung von Beschleunigungsmessungen "in der Praxis" wird dabei auch eines der bislang erfolgreichsten Projekte auf dem Gebiet der parallelen, ereignisgesteuerten Simulation, das sogenannte Time Warp Operating System vorgestellt.

3.5.1 Problematik der Praxis von Beschleunigungsmessungen

Zu Beschleunigungsmessungen an parallelen Simulatoren, seien es nun Prototypen oder in der Praxis eingesetzte Systeme, ist generell zu bemerken, daß solche Messungen nur dann Sinn machen, wenn dabei die Laufzeit des parallelen Simulationslaufs gegen einen echten sequentiellen Simulationslauf gemessen wird. Bei den Meßwerten, die man dagegen in der Literatur findet, wurden vielfach für den sequentiellen Simulationslauf lediglich sämtliche LPs des parallelen Simulators auf einen einzigen Rechnerknoten geladen und dort im Time-Sharing-Verfahren ausgeführt. Hierbei wird der gesamte zusätzliche Aufwand für parallele Simulation bei der Laufzeit des sequentiellen Simulators mitgemessen, was zu völlig unrealistischen, stark überhöhten Beschleunigungswerten führt (vgl. hierzu auch die Ergebnisse in Kapitel 6).

Weiterhin müssen beim sequentiellen Simulator ebenso wie beim parallelen Simulator kritische Zugriffe für die jeweils verwendeten Datenstrukturen optimiert sein. Ein bekanntes Beispiel dafür ist der Zugriff auf die Ereignisliste: Implementiert man sie als doppelt verkettete Liste, so wird die Einfügeoperation bei Ereignislisten, die viele Elemente enthalten, sehr ineffektiv. Bei paralleler Simulation hat jeder Simulator seine eigene (und daher im allgemeinen wesentlich kleinere) Ereignisliste, was die Einfügeoperation erheblich schneller macht. Praktische Erfahrung hat gezeigt, daß es mit diesem "Trick" ohne weiteres möglich ist, superlineare Beschleunigungen zu erzielen.

Schließlich müßte bei den Messungen eigentlich noch berücksichtigt werden, daß das zu einem sequentiellen Simulationsmodell verschmolzene verteilte Simulationsmodell nicht unbedingt die optimale Modellierung des realen Systems darstellt. Beispielsweise kann bei Warteschlangennetzen in einem sequentiellen Simulationsmodell der Abgang eines Kunden an einer Warteschlange und die Ankunft des Kunden an der nächsten Warteschlange als ein *einziges* Ereignis modelliert werden. Bei einem

verteilten Simulationsmodell ist dies im allgemeinen nicht möglich, da die entsprechenden Warteschlangen zu unterschiedlichen Teilmodellen gehören. Nimmt man in diesem Fall für den sequentiellen Simulationslauf die verschmolzene Version des verteilten Simulationsmodells, so hat man doppelt soviele Ereignisse (und damit Ereignislistenzugriffe), als eigentlich notwendig wären.

Insgesamt kann man also sagen, daß Beschleunigungsmessungen, wenn sie zu der Frage "Lohnt sich parallele Simulation?" beitragen sollen, eine sorgfältige Implementierung sowohl des sequentiellen als auch des parallelen Simulators erfordern. Weiterhin ist es wichtig, daß die verwendeten Simulationsmodelle aus der Praxis stammen oder wenigstens nicht gezielt im Hinblick auf "leichte parallele Ausführbarkeit" gestaltet wurden. Veröffentlichungen, die diese Gütekriterien erfüllen, sind selten und es ist häufig auch nicht leicht, nur anhand der veröffentlichten Beschreibung der Implementierung zu entscheiden, wie sorgfältig implementiert und gemessen wurde.

3.5.2 Eine Beispielrealisierung: Das Time Warp Operating System

Um trotz der im vorangegangenen Abschnitt dargelegten Vorbehalte zu zeigen, welche Beschleunigungen nach heutigem Kenntnisstand möglich sind, wird abschließend auf eines der ältesten und ambitioniertesten Projekte auf dem Gebiet der parallelen Simulation eingegangen. Es handelt sich dabei um das *Time Warp Operating System (TWOS)*, das im wesentlichen im Jet Propulsion Laboratory in Zusammenarbeit mit David Jefferson (UCLA) entstand [JBH+85]. Hier wurde im Prinzip ein paralleles Simulationssystem entwickelt, das es gestattet, beliebige Simulationsmodelle mit Hilfe optimistischer Simulationsverfahren auf einem Parallelrechner ablaufen zu lassen.

Aus der Sicht eines Anwenders (d. h. von jemandem, der Simulationsmodelle entwickelt) stellt sich das System im Prinzip wie ein auf parallele Simulation spezialisiertes, verteiltes Betriebssystem dar. Eine Anwendung besteht dabei aus mehreren Objekten (vergleichbar den Prozessen in konventionellen Betriebssystemen), die die Ereignisse ausführen und dabei Ereignisse an andere Objekte verschicken können. Sämtliche Vorgänge, die für die optimistische parallele Simulation notwendig sind, wie etwa das Kopieren der Datenstrukturen, Durchführen von Rollbacks, verzögern irreversibler Operationen etc. werden dabei vom TWOS für den Anwender transparent durchgeführt. Befinden sich mehrere Objekte auf einem Knoten des Parallelrechners, so werden sie nicht wie bei konventionellen Betriebssystemen im Time-Sharing-Verfahren bedient. Statt dessen wird immer das Objekt aktiviert, das von allen auf dem Knoten residierenden Objekten das noch auszuführende Ereig-

nis mit dem kleinsten Zeitstempel besitzt. Weiterhin steht dem Anwender für die Messung sequentieller Simulationsläufe eine spezielle Schnittstelle zur Verfügung, bei der die Ereignisse der Objekte in der gewohnten sequentiellen Weise ausgeführt werden.

Unter TWOS entstanden eine Reihen von Anwendungen, darunter die Computernetzsimulation Warpnet [PEWJ89], die Gefechtsfeldsimulation STB88 [WHF+89] und das aus der Biologie stammende Modell Antopia [EDLP+89], bei dem es um das Futtersuche-Verhalten von Ameisen geht. Beschleunigungen ergaben dabei die folgenden Maximalwerte:

Modell	Anzahl Knoten	Beschleunigung	verwendeter Parallelrechner
Warpnet	32	16.2	JPL/Caltech Mark III Hypercube
STB88	60	28.6	JPL/Caltech Mark III Hypercube
STB88	100	36.8	BBN Butterfly
Antopia	32	12.7	JPL/Caltech Mark III Hypercube

Diese mit nicht-trivialen Modellen (STB88 besteht z. B. aus über 10000 Zeilen C-Code) und vergleichsweise großen Rechnerknotenzahlen erzielten Werte stellen das Ergebnis jahrelanger Arbeiten und Optimierungen am TWOS dar und sind wohl auf Parallelrechnern ohne spezialisiertes Betriebssystem kaum zu erreichen.

Kapitel 4

Warteschlangennetze

In diesem Kapitel wird die spezielle Klasse von Simulationsmodellen, deren parallele Simulation im Rahmen dieser Arbeit untersucht wurde, nämlich *Warteschlangennetze*, vorgestellt. Es handelt sich dabei um Simulationsmodelle, die hauptsächlich bei der Leistungsbewertung von Rechensystemen verwendet werden [Bol89], sich aber auch auf anderen Gebieten, wie etwa der Modellierung von Rechnernetzen [Kle78] einsetzen lassen. Für diese Arbeit wurden sie als Repräsentant einer Vielzahl von Simulationsmodellen ausgewählt, die alle durch folgende Merkmale gekennzeichnet sind:

- Es gibt sogenannte Kunden, die sich in einem Netz von Bedienstationen bewegen.
- Die Bewegung der Kunden sowie ihre Verweilzeiten an den Bedienstationen sind durch Wahrscheinlichkeiten bzw. Zufallsvariablen beschrieben, d. h. insbesondere nichtdeterministisch.
- Sinn der Simulation ist die Bestimmung von Leistungsmerkmalen wie Auslastung von Bedienstationen oder Verweildauer von Kunden im Netz und nicht etwa die Überprüfung der korrekten Funktionsweise des Systems.

Weitere Beispiele für solche Modelle sind Materialflußsysteme oder Straßenverkehrssimulationen. Ein Beispiel, das diese Merkmale nicht aufweist, ist die Simulation von VLSI-Schaltungen.

Der für diese Arbeit verwendete sequentielle Warteschlangennetz-Simulator wurde extra hierfür entwickelt. Dies lag vor allen Dingen daran, daß der Simulator im

Quellcode einer auf Parallelrechnern verfügbaren Programmiersprache (C, Fortan, ...) benötigt wurde. Weiterhin bot die Eigenentwicklung aber auch die Möglichkeit, die Modellwelt so einfach und überschaubar zu halten, daß der Aufwand für die Parallelisierung des Codes eher gering blieb, ohne jedoch die Probleme der Parallelisierung auszusparen. Um praxisnah zu bleiben, wurde daher auch die Beschreibungssprache für die Warteschlangennetze stark an die des Werkzeugs RESQ2 [SMK84] von IBM angelehnt.

Im folgenden werden zunächst Warteschlangennetze allgemein beschrieben. Danach wird die für diese Arbeit verwendete Beschreibungssprache LAVENDER vorgestellt und zum Schluß noch auf die Frage der Darstellung von Warteschlangennetzen als ereignisgesteuerte Simulationsmodelle eingegangen.

4.1 Warteschlangennetze allgemein

Ein *Warteschlangennetz* besteht aus einer (endlichen) Menge von *Warteschlangen*, durch die sich *Kunden* bewegen. Jede Warteschlange besteht dabei aus einem Warteraum und ein oder mehreren Bedienstationen (vgl. Abb. 4.1). Eine einzelne Bedienstation kann (bis auf die weiter unten beschriebenen Ausnahmen) immer nur einen Kunden gleichzeitig bedienen, sie ist somit entweder "belegt" (d. h. sie bedient einen Kunden) oder "frei".

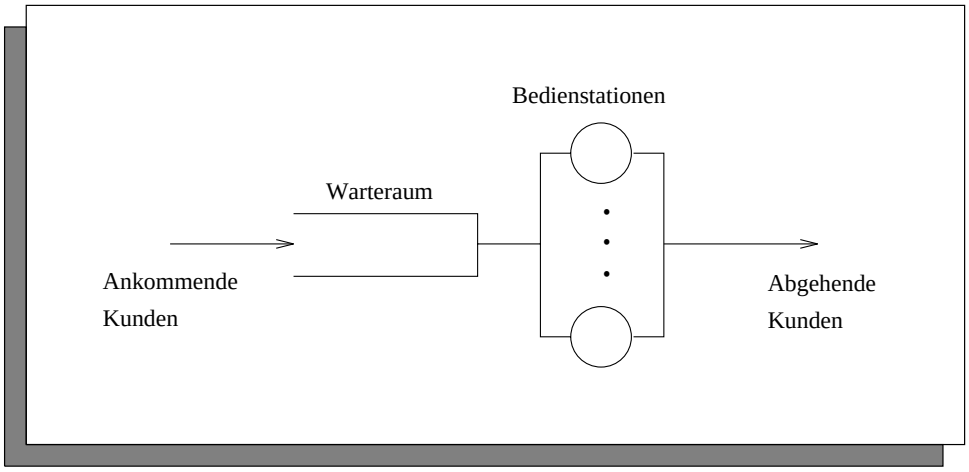


Abbildung 4.1: Warteschlange

Sind bei Ankunft eines Kunden an einer Warteschlange alle Bedienstationen belegt, so begibt sich der Kunde in den Warteraum. Wird eine Bedienstation frei, weil die

Bedienung ihres Kunden beendet ist, und ist der Warteraum nicht leer, so wählt die Bedienstation daraus einen Kunden gemäß der *Bedienstrategie* der Warteschlange aus (z. B. FIFO, d. h. nach der Reihenfolge der Ankunft der Kunden). Die *Bedienzeit* des Kunden, also die Zeit, in der er eine Bedienstation der Warteschlange blockiert, ist eine Zufallsgröße, deren Verhalten durch die *Bedienzeitverteilung* beschrieben wird. Bei Warteschlangen wird somit nur modelliert, wann und wie lange, nicht aber wie ein Kunde bedient wird.

Ist die Bedienung eines Kunden beendet, so verläßt dieser die Warteschlange und wählt eine neue aus, zu der er sich (in Nullzeit) begibt. Die (zufällige) Auswahl wird dabei durch die *Übergangswahrscheinlichkeiten* beschrieben, die für je zwei Warteschlangen i und j angeben, wie groß die Wahrscheinlichkeit ist, daß ein Kunde, der i verläßt, sich nach j begibt.

Ein Warteschlangennetz heißt *offen*, falls es Quellen gibt, die Kunden produzieren können oder falls Kunden aus dem Warteschlangennetz in Senken verschwinden können. In diesem Fall muß dann noch die Verteilung der *Zwischenankunftszeit* der Quelle (d. h. die Zeit, die zwischen der Produktion zweier Kunden durch die entsprechende Quelle vergeht) und die Übergangswahrscheinlichkeiten von den Quellen zu den Warteschlangen und von den Warteschlangen zu den Senke spezifiziert werden. Existieren in dem Warteschlangennetz keine Quellen oder Senken, so nennt man es *geschlossen*.

Eine weitere Verfeinerung von Warteschlangennetzen stellen *Kundenklassen* dar. Hierbei gehört jeder Kunde einer von mehreren Klassen an, wobei die Bedienzeitverteilungen der Warteschlangen und die Übergangswahrscheinlichkeiten je nach Kundenklasse unterschiedlich sein können. Es kann dabei sogar erlaubt sein, daß ein Kunde beim Übergang von einer auf die nächste Warteschlange seine Klasse wechselt. Gibt es für gewisse Kundenklassen des Warteschlangennetzes Quellen oder Senken und für andere Klassen nicht, so heißt das Warteschlangennetz *gemischt*.

Insgesamt ist ein Warteschlangennetz also durch folgendes beschrieben:

- die Kundenklassen
- die Bedienstrategie und die Bedienzeitverteilung der Warteschlangen
- die Übergangswahrscheinlichkeiten zwischen Quellen, Warteschlangen und Senken

4.2 Die Beschreibungssprache LAVENDER

In diesem Abschnitt wird nun die Warteschlangennetz-Beschreibungssprache LAVENDER (**la**nguage for **v**arious qu**e**ueing **n**etwork **de**scr**i**ptions) vorgestellt, in der die Warteschlangennetze für das DISQUE-System (vgl. Kapitel 5) spezifiziert werden. Sie wurde wie bereits erwähnt in Anlehnung an die in RESQ2 [SMK84] verwendete Sprache gestaltet. Statt der genauen Sprachdefinition (die man [Mei91] entnehmen kann) wird hier ein Beispiel ausführlich erläutert, an dem man die wesentlichen Merkmale der Sprache erkennen kann.

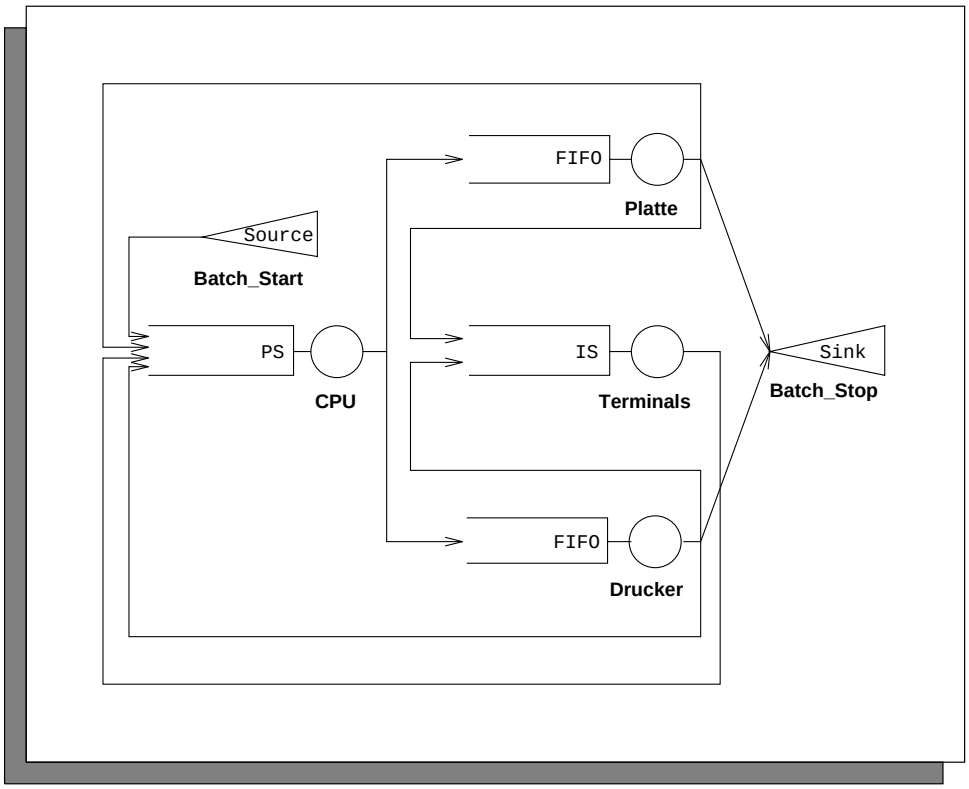


Abbildung 4.2: Großrechner als Warteschlangennetz

Abb. 4.2 zeigt das Warteschlangennetz im Schema. Es handelt sich dabei um ein sogenanntes "Central-Server-Modell", wie sie von Buzen [Buz71] zur Untersuchung des Systemverhaltens von Rechnern mit Multiprogramming vorgeschlagen wurden. Die Idee dabei ist, den Rechner und seine peripheren Geräte (Platte, Drucker und Terminals) durch Warteschlangen zu modellieren. Die Kunden in diesem Netz sind dann gerade die im Time-Sharing-Verfahren bearbeiteten Prozesse. Hierbei werden zwei Sorten unterschieden, die "interaktiven Prozesse", die gelegentlich eine

Eingabe von einem Terminal benötigen und die "Batch-Prozesse", die irgendwo im System entstehen (z. B. automatisch zu bestimmten Zeiten wie etwa regelmäßige Datensicherungen) und terminieren, nachdem sie ihre Aufgabe ausgeführt haben. Die gerichteten Kanten in Abb. 4.2 zeigen dann, wie sich Kunden bewegen können, d. h. zwischen welchen Warteschlangen die Übergangswahrscheinlichkeiten > 0 sind.

Abb. 4.3 zeigt die LAVENDER-Beschreibung dieses Warteschlangennetzes. Sie besteht aus folgenden Teilen (in Klammern das Schlüsselwort, das die Beschreibung einleitet):

- Modellname (model)
- Bedienstrategien und Bedienzeitverteilungen der Warteschlangen (queues)
- Übergangswahrscheinlichkeiten (topology)
- Bei Start der Simulation vorhandene Kunden und Zwischenankunftszeit der Kundenquellen (population)
- Während der Simulation zu bestimmende Leistungsgrößen (statistics)
- Abbruchbedingungen für den Simulationslauf (terminate)

Genauer gilt für die einzelnen Teile folgendes:

Der Modellname dient im wesentlichen dazu, Ausgaben verschiedener Simulationsläufe (z. B. Statistiken) anhand des Namens dem entsprechenden Warteschlangennetz zuzuordnen. Für das Netz selber hat er keine Bedeutung.

Zu der Beschreibung der Warteschlangen muß generell bemerkt werden, daß in Anlehnung an RESQ2 in LAVENDER jede Warteschlange nur eine Bedienstation und eine eigene Menge von Kundenklassen besitzt. Die Bedienzeitverteilung der Kunden der Warteschlange kann für jede Kundenklasse unterschiedlich sein. Als Beispiel schaue man sich die Beschreibung der Warteschlange `CPU` an:

```
queue CPU
        type active [1]
        discipline ps
        classes class iCPU distribution negexp 3.1
                class bCPU distribution negexp 8.2
```

[1] `type active` besagt, daß es sich um eine aktive Warteschlange handelt. Warteschlangen,

```
model Computer_System

queues
        queue Terminals
                type active
                discipline is
                classes class iTerminals distribution negexp 3000.0
        queue CPU
                type active
                discipline ps
                classes class iCPU distribution negexp 3.1
                        class bCPU distribution negexp 8.2
        queue Drucker
                type active
                discipline fifo
                classes class iDrucker distribution normal 200.1 100.6
                        class bDrucker distribution normal 300.6 100.3
        queue Platte
                type active
                discipline fifo
                classes class iPlatte distribution constant 7.0
                        class bPlatte distribution constant 7.0

topology
        from iTerminals to iCPU
        from iCPU to iDrucker iPlatte probabilities 0.1 0.9
        from iPlatte to iCPU iTerminals probabilities 0.7 0.3
        from iDrucker to iCPU iTerminals probabilities 0.7 0.3

        from source to bCPU
        from bCPU to bDrucker bPlatte probabilities 0.3 0.7
        from bDrucker to sink bCPU probabilities 0.8 0.2
        from bPlatte to sink bCPU probabilities 0.8 0.2

population
        at iTerminals 10
        source of bCPU with distribution constant 40000.0

statistics
        at CPU record utilisation
        at Drucker record maxql

terminate
        clock limited to 1000000.0

end
```

Abbildung 4.3: LAVENDER-Beschreibung des Warteschlangennetzes

Dies besagt, daß die Warteschlange `CPU` die beiden Kundenklassen `iCPU` und `bCPU` besitzt (für interaktive und Batch-Prozesse). Die Bedienzeiten der Kunden in den Klassen `iCPU` und `bCPU` sind negativ exponentialverteilt mit den Mittelwerten 3.1 für die Klasse `iCPU` und 8.2 für `bCPU`. Mit `discipline ps` wird die Bedienstrategie der Warteschlange festgelegt. `ps` steht dabei für "Processorsharing", was bedeutet, daß alle in der Warteschlange anwesenden Kunden gleichzeitig bedient werden, die Bedienstation jedoch bei n anwesenden Kunden n-mal solange für die Bedienung jedes Kunden braucht. Man kann sich dies als Grenzfall des Time-Sharing-Verfahrens mit unendlich kleinen Zeitschlitzen vorstellen. Weitere mögliche Bedienstrategien in LAVENDER sind:

- `fifo`: Die Kunden werden in der Reihenfolge ihrer Ankunft bedient.
- `prioq`: (Priority Queue) Die Kundenklassen besitzen unterschiedliche Prioritäten. Wird die Bedienstation frei, so wählt sie unter den anwesenden Kunden mit der höchsten Priorität denjenigen aus, der als erster an der Warteschlange ankam.
- `priopr`: (Priority Queue, Preemptive with Resume) Ist im Prinzip mit `prioq` identisch, nur daß ein Kunde bei Ankunft an der Warteschlange die Bedienung eines Kunden mit niedrigerer Priorität unterbrechen kann. Die Bedienung des unterbrochenen Kunden wird später fortgesetzt.
- `priopnr`: (Priority Queue, Preemtive with No Resume) Ist im Prinzip mit `priopr` identisch, nur daß mit der Bedienung des unterbrochenen Kunden später von vorne angefangen wird.
- `is`: (Infinite Server) Alle anwesenden Kunden werden gleichzeitig und (im Gegensatz zu `ps`) mit unverminderter Geschwindigkeit der Bedienstation bedient.

Die Beschreibung der Übergangswahrscheinlichkeiten geschieht klassenweise,

```
from iCPU to iDrucker iPlatte probabilities 0.1 0.9
```

bedeutet beispielsweise, daß Kunden der Warteschlange `CPU`, die zur Kundenklasse `iCPU` gehören, mit der Wahrscheinlichkeit 0.1 bzw. 0.9 nach dem Ende ihrer Bedienung zu den Warteschlangen `Drucker` bzw. `Platte` gehen und dort den Kundenklassen `iDrucker` bzw. `iPlatte` angehören. Kundenquellen sind in LAVENDER immer einer Klasse einer Warteschlange zugeordnet.

```
from source to bCPU
```

---

so wie sie im vorangegangenen Abschnitt beschrieben wurden, sind immer aktiv. Auf passive Warteschlangen, die der Verwaltung von Ressourcen dienen (vgl. z.B. [SMK84, Mei91]), wird hier nicht näher eingegangen, da sie in dieser Arbeit nicht verwendet werden.

bedeutet, daß es eine Kundenquelle gibt, bei der die von ihr erzeugten Kunden sofort in die Warteschlange `CPU` kommen und dort der Klasse `bCPU` angehören. Der Klassenname `sink` bedeutet, daß der entsprechende Kunde aus dem Warteschlangennetz verschwindet,

```
from bDrucker to sink bCPU probabilities 0.8 0.2
```

heißt also unter anderem, daß Kunden der Klasse `bDrucker` mit einer Wahrscheinlichkeit von 0.8 nach dem Ende ihrer Bedienung aus dem Netz verschwinden.

Auch die Beschreibung der am Start der Simulation vorhandenen Kunden geschieht klassenweise.

```
at iTerminals 10
```

bedeutet, daß sich anfangs in der Warteschlange `Terminals` 10 Kunden befinden, die der Klasse `iTerminals` angehören und alle gerade angekommen sind. Mit

```
source of bCPU with distribution constant 40000.0
```

wird festgelegt, daß die Zwischenankunftzeit der Kundenquelle, die die Klasse `bCPU` speist, konstant 40000 Zeiteinheiten beträgt.

Der Rest der LAVENDER Beschreibung enthält Informationen für den Simulator und hat mit der Beschreibung des Warteschlangennetzes nichts mehr zu tun.

```
at CPU record utilisation
```

besagt, daß für die Warteschlange `CPU` die Auslastung ihrer Bedienstation, also der Anteil an der Gesamtzeit, in der sie belegt war, während des Simulationslaufs bestimmt werden soll. Weitere mögliche Leistungsgrößen sind

`mql`	(mean queue length) Mittlere Anzahl der Kunden im Warteraum der Warteschlange
`maxql`	(maximum queue length) Maximum der Anzahl der Kunden im Warteraum der Warteschlange
`throughput`	Durchsatz (d. h. Anzahl bedienter Kunden pro Zeiteinheit)

Schließlich besagt

```
clock limited to 1000000.0
```

daß der Simulationslauf nach 1000000 simulierten Zeiteinheiten abgebrochen werden soll. Ein anderes Abbruchkriterium ist das Erreichen einer bestimmten Anzahl

bedienter Kunden an einer Warteschlange. Werden mehrere solche Kriterien spezifiziert, so werden sie implizit oder-verknüpft.

Mit Hilfe des Parser-Generators BISON wurde ein Parser für LAVENDER-Dateien entwickelt, der die Warteschlangennetz-Beschreibungen in die intern von den Simulatoren des DISQUE-Testbetts verwendeten Tabellen übersetzt. Sämtliche in dieser Arbeit verwendeten Warteschlangennetze wurden mit LAVENDER spezifiziert.

4.3 Darstellung von Warteschlangennetzen als ereignisgesteuerte Simulationsmodelle

In diesem Abschnitt soll dargestellt werden, wie Warteschlangennetze als ereignisgesteuerte Simulationsmodelle dargestellt werden können, wobei auch auf die unterschiedliche Darstellung als sequentielles und verteiltes Simulationsmodell eingegangen wird. Um dies nicht unnötig technisch und aufwendig zu gestalten, werden dabei die verwendeten Datenstrukturen und die Wirkungsweise der Ereignisse nur informell geschildert.

Der grundlegende Baustein für alle Modelle ist eine Datenstruktur, in der die Kunden, die sich augenblicklich an einer Warteschlange aufhalten, mit ihren angeforderten Bedienzeiten gespeichert sind. Weiterhin müssen für sie die Operationen "Einfügen eines Kunden", "Löschen eines Kunden" sowie die Funktion "Bestimme den Kunden, dessen Bedienzeit abgelaufen ist" vorhanden sein. Diese Datenstruktur wird im folgenden *Kundenraum* genannt.

Modelliert man ein Warteschlangennetz als sequentielles Simulationsmodell, so besteht die Gesamtdatenstruktur des Modells im wesentlichen aus den Daten für die Zufallszahlengeneratoren (vgl. hierzu z. B. [PM88]) und aus je einem Kundenraum für jede Warteschlange. Beim Start der Simulation sind die Kundenräume mit den zu Beginn der Simulation an den jeweiligen Warteschlangen vorhandenen (genauer: gerade angekommenen) Kunden besetzt. Es gibt zwei Sorten Ereignisse:

Übergangsereignisse simulieren die Vorgänge beim Ende der Bedienung eines Kunden. Dieser wird dann aus der Warteschlange, in der er bedient wurde, entfernt. Gemäß der Übergangswahrscheinlichkeiten wird die nächste Warteschlange bestimmt, die er aufsucht. Handelt es sich dabei nicht um eine Senke, so wird der Kunde in den Kundenraum der Warteschlange eingefügt. Ist er der einzige Kunde dort oder unterbricht er die Bedienung eines anderen Kunden (z. B. bei `priopr`), so wird ein Übergangsereignis für das Ende seiner Bedienzeit erzeugt und ggf. das Übergangser-

eignis für den unterbrochenen Kunden gelöscht. Für die alte Warteschlange wird, falls sich noch mindestens ein Kunde in ihrem Kundenraum befindet, ein neues Übergangsereignis für den Zeitpunkt des Endes der Bedienung des nächsten Kunden erzeugt.

Quellenereignisse simulieren die Erzeugung eines neuen Kunden durch eine Kundenquelle. Dazu wird der neu erzeugte Kunde in den Kundenraum der entsprechenden Warteschlange eingefügt und es werden ggf. analog zur Vorgehensweise bei der Ankunft eines Kunden Übergangsereignisse für die Warteschlange gelöscht bzw. erzeugt. Weiterhin wird der Zeitpunkt der Erzeugung des nächsten Kunden gemäß der Verteilung der Zwischenankunftszeit der Quelle bestimmt und ein neues Quellenereignis für diesen Zeitpunkt erzeugt.

Bei Start der Simulation sind für jede Warteschlange mit nichtleerem Kundenraum ein Übergangsereignis und für jede Quelle ein Quellenereignis mit den entsprechenden Zeitstempeln vorhanden.

Will man ein Warteschlangennetz als verteiltes Simulationsmodell darstellen, so stellt sich zunächst die Frage, wie es in Teilmodelle zerlegt werden soll. Es bietet sich dabei an, jede Warteschlange als einzelnes Teilmodell darzustellen, da man damit das Netz in die "kleinsten Einheiten" zerlegt hat und mit Hilfe des Verschmelzens (vgl. 2.3) sehr flexibel beim Erzeugen größerer Teilmodelle ist. Dies hat zur Folge, daß der Übergang eines Kunden von einer Warteschlange in die nächste nicht mehr als ein einzelnes Ereignis dargestellt werden kann, da ein Ereignis nicht die Datenstrukturen zweier Teilmodelle verändern kann. Daher wird hier das Übergangsereignis durch zwei Ereignisse, das *Abgangsereignis* und das *Ankunftsereignis* ersetzt: Das Abgangsereignis simuliert den Abgang eines Kunden, dessen Bedienung beendet ist (d. h. Entfernen des Kunden aus dem Kundenraum, ggf. neues Abgangsereignis erzeugen, nächste Warteschlange bestimmen). Darüber hinaus erzeugt es ein Ankunftsereignis für die Warteschlange, zu der der Kunde sich als nächstes begibt, falls er nicht in einer Senke verschwindet. Entsprechend simuliert das Ankunftsereignis die Ankunft eines Kunden an einer Warteschlange (d. h. Kunden in Kundenraum einfügen, ggf. altes Abgangsereignis löschen, ggf. neues Abgangsereignis erzeugen). Da Quellenereignisse immer nur den Zustand einer einzigen Warteschlange verändern, können sie aus dem sequentiellen Simulationsmodell direkt übernommen werden. Analog zum sequentiellen Simulationsmodell sind beim Start der Simulation ein Abgangsereignis für jede Warteschlange mit nichtleerem Kundenraum und ein Quellenereignis für jede Quelle vorhanden.

Kapitel 5

Beschreibung des Testbetts DISQUE

Dieses Kapitel beschreibt das Testbett DISQUE (**Di**stributed **S**imulator for **Que**ueingnetworks), mit dem im Rahmen dieser Arbeit Experimente zur Analyse des Beschleunigungsverhaltens durchgeführt wurden. Dazu werden zunächst die verwendeten Rechner und Softwaresysteme vorgestellt und die mit ihnen gemachten Erfahrungen diskutiert. Danach werden die verschiedenen Simulatoren, mit denen die Experimente durchgeführt wurden, beschrieben. Schließlich wird noch auf die Werkzeuge zur Leistungsvorhersage und zur Leistungsanalyse eingegangen. Eine kurze Übersicht über das gesamte System findet man in [Ric91].

5.1 Beschreibung der verwendeten Parallelrechner

Da sich der ursprünglich für diese Arbeit verwendete Parallelrechner als zu ineffektiv und zu instabil erwies, ergab sich bereits während der Entwicklungsphase von DISQUE die Notwendigkeit, die Software auf andere Rechner zu portieren. Dies führte zwar zu einem erheblichen zeitlichen Mehraufwand bei der Programmentwicklung, hatte andererseits aber vor allen Dingen Vorteile beim Austesten der Software, da dafür die komfortabelste und nicht die effektivste Programmierumgebung verwendet werden konnte. Weiterhin bestand so die Möglichkeit, das Laufzeitverhalten paralleler Simulationsläufe auf Parallelrechnern und Workstationnetzen miteinander zu vergleichen.

5.1.1 Parsytec Multicluster 2

Das Parsytec-Multicluster-System basiert auf dem Prozessor T800 der Firma INMOS, der zur Prozessorfamilie der Transputer gehört. Die Prozessoren dieser Familie zeichnen sich alle dadurch aus, daß sich bei ihnen auf dem Chip bereits die Hardware für einen Scheduler für leichtgewichtige Prozesse sowie für vier bidirektionale Kanäle zur Kommunikation mit anderen Transputern, die sogenannten Links, befinden. Auf diese Weise ist die Kommunikation zwischen Transputern einfach und ohne zusätzliche Hardware möglich. Beim Multicluster 2 (Abb. 5.1) werden die Links über die sogenannte NCU (Network Control Unit) miteinander verbunden. Hierbei handelt es sich um einen konfigurierbaren Crossbarswitch, der es ermöglicht, Transputernetze mit beliebigen Topologien zu schalten (natürlich nur unter Berücksichtigung der beschränkten Anzahl von Links je Transputer). Für die Kommunikation mit dem Host (in diesem Fall eine SUN 3/140 mit VME-Bus) gibt es ein spezielles Einschubboard. Auf ihm befinden sich vier Transputer (die sogenannten Root-Transputer), bei denen jeweils ein Link über einen Adapter mit dem VME-Bus und ein weiterer Link mit der NCU verbunden ist.

Als Betriebssystem für den Multicluster 2 wurde Helios [Ltd90] verwendet. Jeder Helios-Benutzer benötigt einen Root-Transputer exklusiv. Auf ihm werden mehrere leichtgewichtige Prozesse gestartet, die im wesentlichen für die Kommunikation zwischen dem Host und den Transputern des Clusters (Zugriff auf Dateien des Hosts, Bildschirm-Ein/Ausgabe etc.) zuständig sind. Außerdem wird dort für den Benutzer eine Shell gestartet, mit der er Transputer aus dem Cluster allokieren und verschalten sowie anschließend seine Anwendung laden und starten kann.

Dieses System stellte die ursprüngliche Entwicklungsumgebung für das Testbett DISQUE (vgl. 5.2) dar. Benutzt wurden dabei die Programmiersprache C und das Programmiermodell von Helios. Letzteres stellt im wesentlichen eine UNIX-ähnlichen Bibliothek von E/A-Funktionen, Primitive zum Erzeugen leichtgewichtiger Prozesse sowie Funktionen zur Kommunikation zwischen diesen über sogenannte Pipes zur Verfügung. Hierbei handelt es sich um bidirektionale Kanäle, die aus der Sicht des Anwendungsprogramms wie normale UNIX-Dateien behandelt werden. Insbesondere können sie auch über globale (d. h. auf allen Rechnerknoten bekannte) Namen von beliebigen Prozessen beschrieben oder gelesen werden. Die Synchronisation von Prozessen eines Rechnerknotens erfolgt unter Helios über Semaphore.

Ursprünglich war geplant, die gesamte Arbeit ausschließlich auf dem Multicluster 2 durchzuführen. Jedoch erwies sich das System dafür als zu instabil und zu unzuverlässig. Neben der schlechten Qualität der Helios-Software machte sich vor allem auch das Fehlen jeglicher Speicherschutzmechanismen bei der Hardware des Trans-

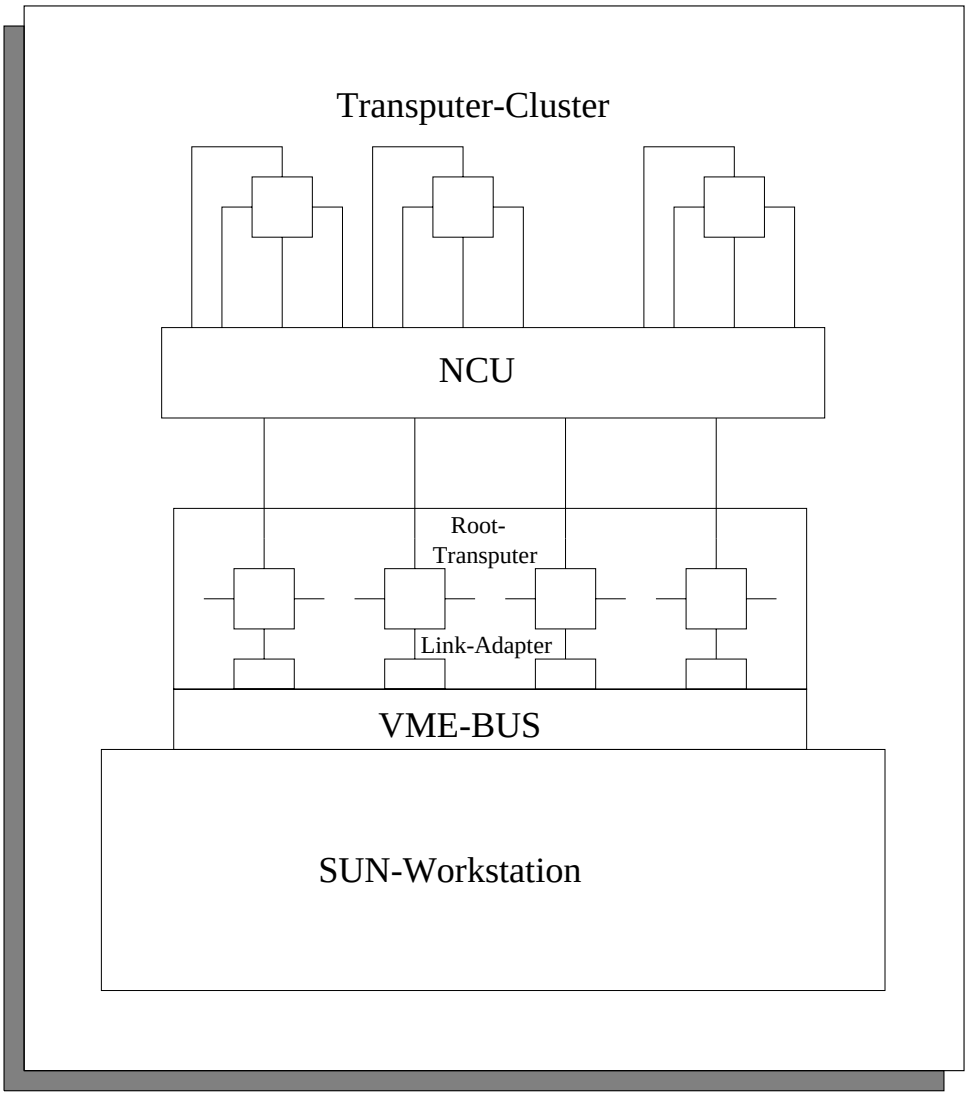


Abbildung 5.1: Parsytec Multicluster 2

puters bemerkbar, was gerade bei der Programmierung in C leicht zum Zerstören von Daten des Betriebssystems durch fehlerhafte Programme führen kann. Da sich außerdem noch gewisse Funktionen von Helios als äußerst ineffektiv erwiesen (beispielsweise hat das Schreiben auf Helios-Pipes eine Startzeit von ca. 2 Millisekunden, was dem mehr als zwanzigfachen der reinen Hardware-Startzeit für Transputerlinks entspricht), wurde im Verlauf dieser Arbeit die Software auf die weiter unten beschriebenen Systeme portiert und der Multicluster 2 nicht mehr weiter benutzt.

5.1.2 Emulation unter UNIX

Hierbei handelt es sich nicht um einen Parallelrechner, sondern um eine Parallelrechner-Emulation auf einer UNIX-Workstation. Sie entstand aus der Notwendigkeit, eine zuverlässige und komfortable Programmierumgebung zu haben, nachdem sich Helios immer mehr als Hindernis bei der Programmentwicklung erwies.

Entwickelt wurde sie auf einer SUN Sparcstation 2 unter SUNOS 4.1.3 mit Hilfe der (SUN-spezifischen) Lightweightlibrary. Diese ermöglicht es, innerhalb eines normalen UNIX-Prozesses mehrere leichtgewichtige Prozesse (sogenannte LWPs) ablaufen zu lassen, die alle denselben Adreßraum (nämlich den des UNIX-Prozesses) besitzen. Damit war es möglich, ein Transputersystem unter Helios zu emulieren, indem man jeden Transputer durch einen UNIX-Prozeß und jeden Prozeß auf dem Transputer durch einen LWP ersetzte. Da in Helios viele Funktionen direkt aus UNIX übernommen wurden (z. B. Bildschirm-Ein/Ausgabe, Dateizugriffe) und wenige Helios-Funktionen von den Komponenten von DISQUE benutzt wurden, war es mit vertretbarem Aufwand möglich, die Helios-spezifischen Teile von DISQUE unter UNIX nachzuprogrammieren. Ein besonderes Problem war dabei noch die Emulation der Kommunikation über Helios-Pipes. Da diese mit einem auf allen Rechnerknoten identischen Namen erzeugt und angesprochen werden, waren normale UNIX-Pipes nicht verwendbar, da sie nur neu erzeugten Prozessen vom erzeugenden Prozeß übergeben werden können und somit zwei Prozesse nicht nachträglich eine Pipe zur Kommunikation miteinander einrichten können. Eine elegante Lösung stellten die unter SUNOS ebenfalls vorhandenen Named Pipes dar, die sich zwar genau wie UNIX-Pipes verhalten, aber Objekte im UNIX-Dateibaum sind und über die entsprechenden Pfadnamen angesprochen werden. Sie ließen sich daher im Prinzip genau wie Helios-Pipes benutzen.

Für die Entwicklung des DISQUE-Testbetts erwies sich die Emulation als großer Fortschritt. Dies hatte mehrere Gründe: Da eine UNIX-Workstation ein erheblich stärker ausgereiftes und damit vor allen Dingen zuverlässigeres System darstellt, wurde die Entwicklung nicht ständig durch Systemzusammenbrüche und an-

schließende Neustarts behindert. Weiterhin besitzt der Sparc-Prozessor Speicherschutzmechanismen, mit denen man die gerade bei der Programmierung in C häufig auftretenden illegalen Speicherzugriffe sofort erkennen kann. Außerdem erwiesen sich die Vielzahl der unter UNIX vorhandenen Werkzeuge, insbesondere erprobte Debugger wie etwa gdb, als große Hilfe.

Zusammenfassend kann man sagen, daß auch für zukünftige Software-Projekte für Parallelrechner der Gebrauch eines Emulators auf einer Workstation äußerst empfehlenswert ist, insbesondere in der Testphase neuer Software, wenn die Mehrheit der Fehler noch im sequentiellen Code steckt und nichts mit den speziellen Problemen der Parallelverarbeitung zu tun hat.

5.1.3 Intel iPSC860-Hypercube

Alle Messungen dieser Arbeit wurden auf dem iPSC860-Hypercube der Firma Intel durchgeführt. Abb. 5.2 zeigt die Komponenten des Systems.

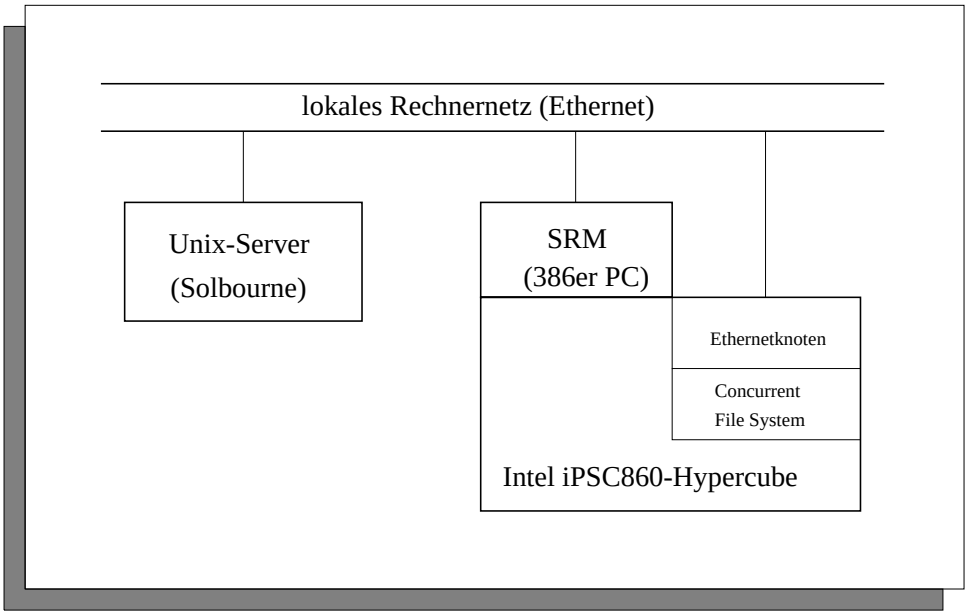


Abbildung 5.2: Intel iPSC860-Hypercube

Der iPSC860 besteht im wesentlichen aus 32 Rechnerknoten, die mit Prozessoren vom Typ i860 der Firma Intel und einem Kommunikationsprozessor, dem DCM (Direct Connect Module) bestückt sind. Über die DCM sind die Rechnerknoten

in Hyperwürfel-Topologie miteinander verbunden. In diese Topologie sind außerdem noch weitere Rechnerknoten eingebunden, die für gewisse Dienstleistungen zur Verfügung stehen:

Die sieben Knoten des CFS (Concurrent File System) bilden ein paralleles Dateisystem, das einen schnellen Dateizugriff für lokale Anwendungen ermöglicht. Jeder Knoten des CFS ist für den Zugriff auf eine Platte zuständig, wobei die vom CFS verwalteten Dateien in Blöcken von 4k Größe auf die Platten verteilt werden. Auf diese Weise ist ein paralleler Dateizugriff von den Rechnerknoten einer Anwendung aus möglich.

Ein weiterer Serviceknoten betreibt einen Ethernet-Controller, der an ein lokales Rechnernetz (in diesem Fall das Campus-Netz der Universität) angeschlossen ist. Dieser wird im wesentlichen für zwei Aufgaben eingesetzt: Einmal ermöglicht er einen effektiven Transfer von Dateien zwischen dem CFS und dem Dateisystem eines anderen an das Rechnernetz angeschlossenen Rechners über FTP (File Transfer Protocol [PR85]). Zum anderen stellt er den auf den Rechnerknoten laufenden Anwendungen die aus dem Berkeley-Unix stammende Socket-Schnittstelle zur Verfügung, die diesen eine effiziente Kommunikation mit anderen Rechnern des Rechnernetzes ermöglicht.

Als Frontend-Rechner für den iPSC860 dient ein PC mit einem 80386 Prozessor und UNIX (SV/R4) als Betriebssystem. Dieser sogenannte SRM (System Resource Manager) ist für die Verwaltung der Resourcen des iPSC860 (d. h. im wesentlichen für das Allokieren und Freigeben von Rechnerknoten für einzelne Benutzer) sowie das Laden der Anwendungen auf die Rechnerknoten zuständig. Weiterhin kann dort für jede Anwendung eine sogenannte Hosttask gestartet werden. Dies ist ein UNIX-Prozeß, der in der Lage ist, Nachrichten an einzelne Rechnerknoten, auf denen die Anwendung läuft, zu schicken oder von dort zu empfangen. Außerdem ist es seine Aufgabe, Bildschirm-Ein/Ausgaben der auf den Knoten des iPSC860 laufenden Anwendungen an deren Benutzer weiterzureichen.

Da diese Aufgaben einen PC (noch dazu mit einem 80386 Prozessor) im allgemeinen etwas überfordern, wurden sie teilweise auf einen UNIX-Server (in diesem Fall eine Solbourne-Maschine) ausgelagert (Remote Hosting). Dabei kann ein Anwender auf dem Server genauso wie auf dem SRM Knoten anfordern und Anwendungen laden. Der Server leitet dann die entsprechenden Befehle an den PC bzw. die Antworten von dort an den Benutzer weiter, hat also keinen direkten Zugriff auf den iPSC860. Weiterhin laufen auf ihr die Crosscompiler für C und FORTRAN, mit denen der Objektcode für die i860-Prozessoren erzeugt wird.

Das Hauptproblem bei der Portierung der Software des DISQUE-Testbetts auf den

iPSC860 war, daß das dort vorhandene Betriebssystem NX2 nur einen Prozeß pro Rechnerknoten erlaubt. Abhilfe schaffte hier der Betriebssystemkern MMK der an der Universität München entwickelten Programmierumgebung TOPSYS [BBLT90], der aufbauend auf NX2 auch auf dem iPSC860 läuft. Er stellte außer leichtgewichtigen Prozessen auch gleich noch die Lösung eines weiteren Problems zur Verfügung: Die Prozesse unter MMK kommunizieren über Mailboxen miteinander, die zur Laufzeit der Anwendung erzeugt werden können und mit auf allen Rechnerknoten bekannten Identifikatoren angesprochen werden. Somit konnten die Helios-Pipes auf dem iPSC860 durch eben diese Mailboxen ersetzt werden.

Schließlich stellte noch die extrem langsame Kommunikation zwischen dem iPSC860 und dem SRM ein Problem dar, insbesondere was die beim Debuggen anfallenden größeren Mengen von Bildschirmausgaben betraf. Hier ließ sich der Ethernetknoten des iPSC860 einsetzen. Mit Hilfe der durch ihn zur Verfügung gestellten Socketschnittstelle wurden die Bildschirmausgaben der Rechnerknoten über das lokale Rechnernetz zu einem Prozeß auf einer UNIX-Workstation umgeleitet, der die Ausgaben nach Rechnerknoten getrennt in verschiedenen Fenstern der Workstation darstellte.

Insgesamt kann man feststellen, daß der iPSC860 zwar in Stabilität, Effizienz und Programmierkomfort immer noch nicht mit einer UNIX-Workstation konkurrieren kann, jedoch in diesen Punkten gegenüber dem Parsytec-System eine erhebliche Verbesserung darstellt.

5.1.4 Workstation-Netz

Diese Portierung stellt ein "Abfallprodukt" der Portierungsarbeiten für den iPSC860 dar. Der Grund dafür ist, daß der dort benutzte MMK-Kern auch auf UNIX-Workstations mit SPARC-Prozessoren implementiert wurde. Somit lief die iPSC860 Portierung von DISQUE auch fast ohne Änderungen auf über Ethernet vernetzten SUN-Workstations vom Typ ELC unter SUNOS 4.1.3.

Im Rahmen dieser Arbeit wurde dieses Workstationnetz im wesentlichen für Debugging-Zwecke eingesetzt. Es erwies sich dabei insbesondere deswegen als sehr nützlich, da es sich beim MMK-Kern selber noch um eine Beta-Version handelte und der Debugger des iPSC860 nicht zusammen mit der Remote-Hosting-Software einsetzbar war. Für Messungen an parallelen Simulationsläufen wurde es nicht eingesetzt, da das Ethernet und die Workstations ständig auch für andere Zwecke benötigt wurden und störungsfreie Experimente daher nur schwer möglich waren.

5.2 Simulatoren

5.2.1 Sequentielle Simulatoren

Wie bereits in 4.3 näher erläutert wurde, kann man ein Warteschlangennetz auf unterschiedliche Arten als ereignisgesteuertes Simulationsmodell darstellen. Entsprechend gibt es im DISQUE-Testbett zwei verschiedene sequentielle Simulatoren, `optsim` und `simulate`. Bei `optsim` ist das verwendete Simulationsmodell für die sequentielle Simulation optimiert, indem der Abgang eines Kunden und seine Ankunft an der nächsten Warteschlange als ein einziges Ereignis modelliert wird. Bei `simulate` sind dagegen genau wie bei den verteilten Simulatoren Abgang und Ankunft eines Kunden zwei verschiedene Ereignisse.

In der Handhabung sind beide Simulatoren identisch: Man übergibt ihnen ein Warteschlangennetz in Form einer LAVENDER-Beschreibung, die sie in ein internes Format umwandeln, anschließend den Simulationslauf der gewünschten Länge durchführen und schließlich die gewünschten Statistiken ausgeben. Abb. 5.3 zeigt die (aus Platzgründen leicht gekürzte) Ausgabe von `simulate` für das in Abb. 4.3 dargestellte Beispiel. Darüber hinaus kann man durch zusätzliche Parameter die Simulatoren veranlassen, noch weitere Informationen auszugeben oder als Datei anzulegen (z. B. Orakeldateien, vgl. 5.3.2). Hierauf wird noch später eingegangen.

Im Rahmen dieser Arbeit wurden die beiden Simulatoren im wesentlichen dazu benutzt, die Grundlage für Beschleunigungsmessungen zu liefern. Dabei zeigt der Vergleich der Laufzeit von `optsim` und `simulate`, wie stark sich der Aufwand für die Simulation durch die Anpassung des Simulationsmodells an die besonderen Gegebenheiten von Parallelrechnern erhöht, d. h. wie groß die Mehrarbeit ist, die ein paralleler Simulator wegen des inhärent ungünstigeren Simulationsmodells zu leisten hat. Der Vergleich der Laufzeiten von `simulate` und einem parallelen Simulator zeigt dagegen, wie stark die parallele Simulation den Simulationslauf wirklich beschleunigt. Schließlich ergibt sich die Beschleunigung, die einen Anwender interessiert, der lediglich seine Anwendung schneller laufen lassen will, aus dem Vergleich der Laufzeiten von `optsim` und einem parallelen Simulator.

5.2.2 Parallele Simulatoren

Die parallelen Simulatoren des DISQUE-Systems benutzen dieselbe Beschreibung der Warteschlangennetze wie die sequentiellen Simulatoren. Da es aber bei ihnen pro Rechnerknoten nur einen einzigen Simulator gibt, benötigen sie noch zusätzli-

```
Ergebnisse des Simulationslaufs fuer Testbeispiel : Computer_System
---------------------------------------------------------------------------
Zeit bei Simulationsende : 1000000.00000
---------------------------------------------------------------------------
Mittlere Warteschlangenlaengen :
---------------------------------------------------------------------------
Maximale Warteschlangenlaengen :
             Drucker :    5.00000
            iDrucker :    5.00000
            bDrucker :    1.00000
---------------------------------------------------------------------------
Maximale Wartezeiten :
---------------------------------------------------------------------------
Durchsatz :
---------------------------------------------------------------------------
Auslastung der Server :
                 CPU :    3.29598 %
                iCPU :    3.27039 %
                bCPU :    0.02559 %
---------------------------------------------------------------------------
Anzahl bedienter Jobs :
           Terminals :            3205
          iTerminals :            3205

                 CPU :           10532
                iCPU :           10502
                bCPU :              30

             Drucker :            1103
            iDrucker :            1094
            bDrucker :               9

              Platte :            9429
             iPlatte :            9408
             bPlatte :              21
---------------------------------------------------------------------------
Anzahl der aktiven und/oder wartenden Jobs bei Simulationsende :
                                          wartend    aktiv

           Terminals :              10         0        10
          iTerminals :              10         0        10

                 CPU :               0         0         0
                iCPU :               0         0         0
                bCPU :               0         0         0

             Drucker :               0         0         0
            iDrucker :               0         0         0
            bDrucker :               0         0         0

              Platte :               0         0         0
             iPlatte :               0         0         0
             bPlatte :               0         0         0
---------------------------------------------------------------------------

*****************************************************************************
*                                                                           *
*      Zeitmessungen:                                                       *
*      Initialisieren                    :           9715 micro sec         *
*      Parser                            :          54065 micro sec         *
*      Simulator                         :        3776761 micro sec         *
*      Simulator ohne Ergebnisausgabe    :        3688970 micro sec         *
*      Gesamtdauer                       :        3846723 micro sec         *
*                                                                           *
*****************************************************************************
```

Abbildung 5.3: Von `simulate` erzeugte Ausgabe

che Informationen darüber, wie die Warteschlangen (aufgefaßt als Teilmodelle) zu der entsprechenden Anzahl von Teilmodellen zu verschmelzen sind. Im DISQUE-Testbett gibt es dabei zwei verschiedene Möglichkeiten: Einmal kann man die Verteilung der Warteschlangen auf die Rechnerknoten direkt in Form einer Datei angeben, in der zu jeder Warteschlange die Rechnerknotennummer "ihres" Simulators angegeben ist. Diese Datei muß dann allerdings "von Hand" erstellt werden, was nur bei einfachen, überschaubaren Warteschlangennetzen möglich ist. Die andere Möglichkeit besteht darin, den Simulator selber die Verteilung vornehmen zu lassen. In DISQUE wurden dazu zwei Verfahren basierend auf den Graphpartitionierungsalgorithmen von Kerningham/Lin [KL70] und SOCCER [Run88] sowie ein Verfahren, das zu zyklenfreier Kommunikation zwischen den Simulatoren der Teilmodelle führt, implementiert. Auf letzteres Verfahren wird in Kapitel 6 noch näher eingegangen, die beiden erstgenannten Verfahren wurden für die Beschleunigungsmessungen im Rahmen dieser Arbeit nicht benutzt.

Die Ergebnisausgaben der parallelen Simulatoren unterscheiden sich nicht wesentlich von denen der sequentiellen. Aus Gründen der einfacheren Implementierung gibt jedoch jeder Rechnerknoten die Statistiken der auf ihm simulierten Warteschlangen getrennt aus.

Gestartet wird der gesamte Simulationslauf von einem Rechnerknoten, der sich ausschließlich um den Ablauf der Simulation kümmert und auf dem daher kein Simulator läuft. Auf diese Art wird zwar ein Rechnerknoten vergeudet, dies war aber in Anbetracht der Tatsache, daß einige Aufgaben der Ablaufsteuerung zeitkritisch sind (z. B. die Uhrensynchronisation bei der Erzeugung von Tracefiles, vgl. 5.3.3) leider notwendig. Die Ablaufsteuerung startet auf allen anderen Rechnerknoten einen Prozeß, den *Hauptprozeß*, der für die Kommunikation mit der Ablaufsteuerung zuständig ist. Anschließend nimmt sie die Zuordnung der Warteschlangen zu den einzelnen Rechnerknoten vor (durch den vom Benutzer ausgewählten Algorithmus oder durch Einlesen der entsprechenden Datei) und sendet die Daten der Warteschlangen an die entsprechenden Hauptprozesse. Diese legen sie dort im Speicher ab, generieren die Ereignisliste sowie die beim Start der Simulation initial vorhandenen Ereignisse und starten eine Reihe weiterer Prozesse (vgl. Abb. 5.4): Der *Simulatorprozeß* führt die eigentliche Simulation durch, indem er in einer Endlosschleife immer das vorderste Ereignis der Ereignisliste ausführt. Ist die Ereignisliste leer, so wartet er am Semaphor `new_job_arrival` auf die Ankunft neuer Ereignisse. Dort wird er von den *Leseprozessen* aktiviert, die die auf dem Knoten eintreffenden Ereignisnachrichten entgegennehmen. Jeder Leseprozeß ist dabei für die Ereignisnachrichten eines Rechnerknotens zuständig. Dies war notwendig, da es unter Helios nicht möglich ist, mit einem Prozeß auf Daten aus mehreren Pipes zu warten (wie z. B. unter UNIX mit der `select` Funktion), andererseits aber auch auf

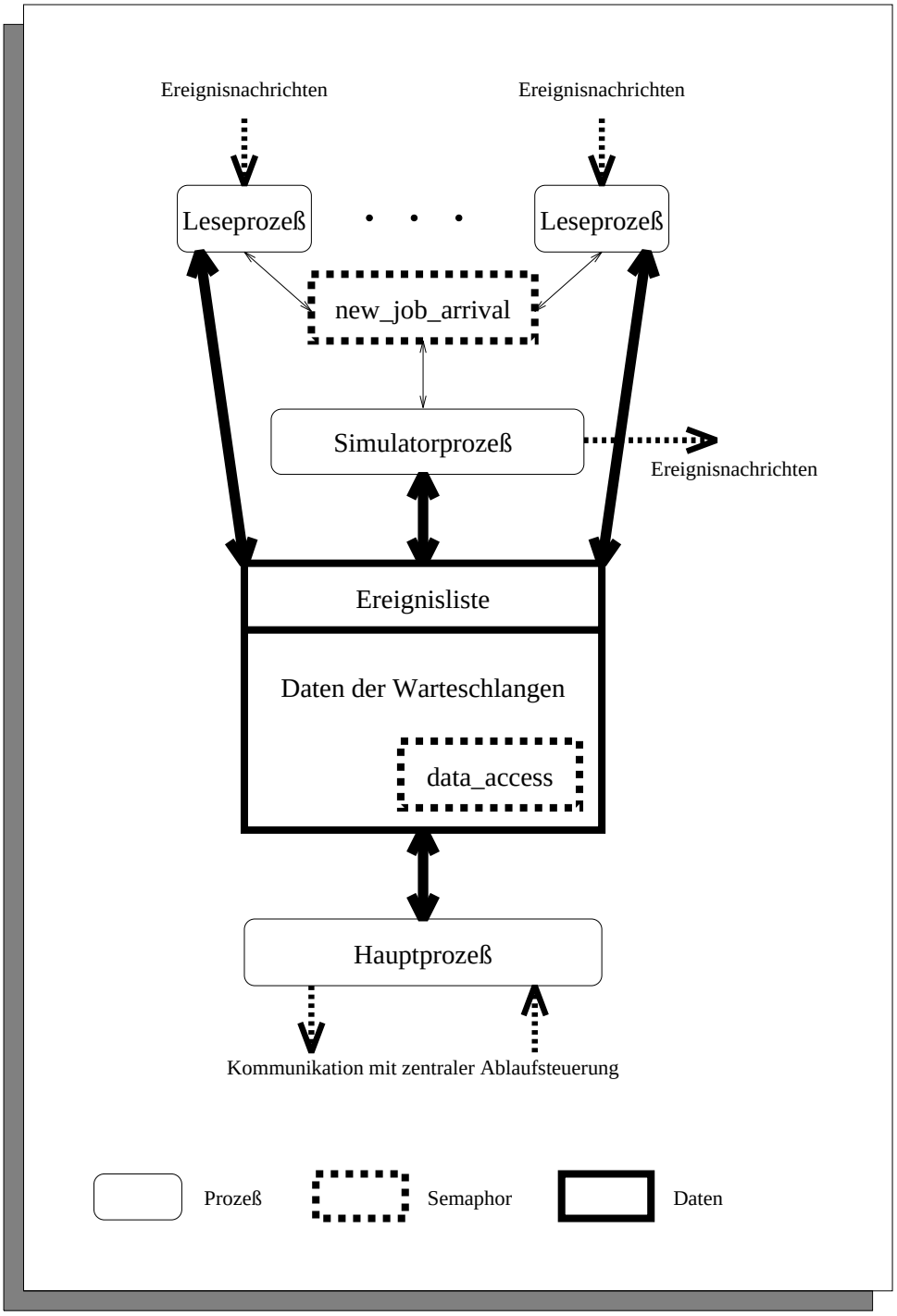


Abbildung 5.4: Prozeßstruktur eines Simulatorknotens

eine Helios-Pipe nicht mehrere Prozesse gleichzeitig schreiben können, ohne daß die Nachrichten durcheinander geraten und regelmäßiges Pollen aller Pipes wegen der bereits erwähnten hohen Zugriffszeiten ausschied. Somit stellt dies auch ein gutes Beispiel dafür dar, wie ein fehlendes Betriebssystemkonzept die gesamte Struktur einer Anwendung negativ beeinflussen kann.

Da sämtliche Prozesse eines Simulatorknotens auf gemeinsame Daten (Ereignisliste und Daten der Warteschlangen) zugreifen, muß dieser Zugriff durch einen Semaphor geschützt werden. Dies führt unter anderem dazu, daß vor und nach der Ausführung eines Ereignisses durch den Simulatorprozeß eine Semaphor-Operation durchzuführen ist, was, wie sich später noch zeigen wird, bedingt durch die feine Granularität der Anwendung zu erheblichen Leistungseinbußen führt. Dies zeigt recht deutlich, daß bei der Verwendung von Multithreading nicht nur Kosten für den Prozeßwechsel (die z. B. bei Transputern sehr niedrig sind) sondern auch für die Synchronisation zwischen Prozessen anfallen, so daß die hier verwendete Prozeßstruktur ein durch das Konzept des Transputers (Hardware-Scheduler für Threads) inspirierter Fehler war.

Bei der Realisierung unterschiedlicher Synchronisationsverfahren wurde das Filterkonzept von Reynolds [RD89] verwendet. Ausgehend von der Idee, daß ein Großteil des Codes eines parallelen Simulators unabhängig vom versendeten Synchronisationsverfahren ist, wird dabei der Code, für den das nicht gilt, in Form sogenannter Filter konzentriert. Im parallelen Simulator des DISQUE-Testbetts werden folgende Filter verwendet:

Dem *in_filter* werden von den Leseprozessen die eingegangenen Nachrichten übergeben. Er führt alle Operationen, die beim Eintreffen einer Nachricht nötig sind, aus. Beispiele dafür sind das Einfügen von Ereignissen in die Ereignisliste, das Ändern der Kanalzeiten beim Eintreffen von Nullnachrichten (die dann nicht in die Ereignisliste eingefügt und damit "herausgefiltert" werden) und das Auslösen von Rollbacks beim Eintreffen von Stragglern.

Der *sim_filter* wird vom Simulatorprozeß vor der Ausführung jedes Ereignisses aufgerufen. Er überprüft, ob dem Prozeß die Erlaubnis zur Ausführung des Ereignisses erteilt werden darf (was nur bei konservativen Verfahren eine Rolle spielt, bei optimistischen Verfahren wird sie immer erteilt) und führt ggf. die Aktionen durch, die vor der Ausführung eines Ereignisses stattfinden müssen (z. B. Speichern einer Zustandskopie bei optimistischen Verfahren). Wird dem Simulatorprozeß die Erlaubnis verweigert, so wartet er wie bei einer leeren Ereignisliste am Semaphor `new_job_arrival` auf die nächste Nachricht.

Schließlich gibt es noch den *out_filter*. Er wird vom Simulatorprozeß für jede von ihm

erzeugte Ereignisnachricht aufgerufen. Seine Aufgabe ist es, die Nachricht an den entsprechenden Rechnerknoten zu verschicken und die vorher notwendigen Aktionen durchzuführen (z. B. Verschicken von Nullnachrichten, Kopieren der Nachricht für die Liste der Antinachrichten).

Darüber hinaus gibt es noch Filter, die bei Start und Ende des Simulationslaufs aufgerufen werden und eher technische Aufgaben haben (z. B. Öffnen und Schließen von Pipes). Zu erwähnen wäre noch, daß für die optimistischen Verfahren auch einige Stellen des Codes außerhalb der Filter modifiziert werden mußten, z. B. bei der Freigabe von Speicher nach der Ausführung eines Ereignisses.

Der Einsatz des Filterkonzepts erwies sich als voller Erfolg, da es dadurch möglich war, Personen, die nur eine grobe Vorstellung von der Funktionsweise des parallelen Simulators besaßen, zum Realisieren unterschiedlicher Synchronisationsverfahren heranzuziehen. Dies war vor allen Dingen deswegen vorteilhaft, da viele Arbeiten am Testbett in Form von Projekt- und Diplomarbeiten [Mei91, Sch91, Mei92, Win93, Sch93, Mül93] von verschiedenen Studenten durchgeführt wurden.

5.3 Leistungsvorhersage / Leistungsanalyse

5.3.1 Kritische-Pfad-Analyse

Bei `cpasim` handelt es sich um eine Variante des Warteschlangennetz-Simulators `simulate`, die mit den in 3.4.1 vorgestellten Algorithmen zur Analyse der Lastbalancierung und des kritischen Pfades instrumentiert wurden. Genau wie `simulate` liest auch `cpasim` das Warteschlangennetz in Form einer LAVENDER-Beschreibung ein und simuliert es. Anschließend werden jedoch nicht nur die gewünschten Statistiken, sondern auch die Ergebnisse der Analyse der Lastbalancierung und der Kritischen-Pfad-Analyse ausgegeben.

Bei diesen Analysen wird davon ausgegangen, daß jede Warteschlange von einem eigenen Prozessor simuliert wird. Alternativ dazu kann aber auch eine Datei eingelesen werden, in der (im selben Format wie bei den parallelen Simulatoren) die Zuordnung von Warteschlangen zu Rechnerknoten beschrieben ist. Weiterhin ist es möglich, statt von einer Übertragung der Ereignisnachrichten in Nullzeit auszugehen, dafür eine (konstante) Zeit als Parameter anzugeben. Dies stellt eine einfache Erweiterung des Algorithmus aus 3.4.1 dar.

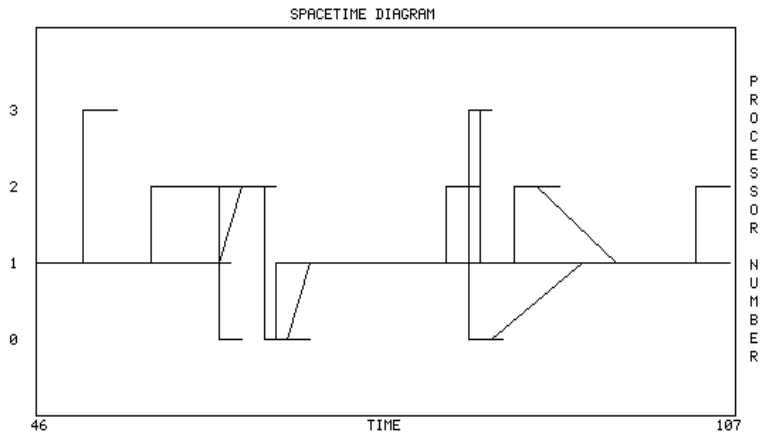
Schließlich ist es auch möglich, sich den Verlauf der idealisierten parallelen Simu-

lation, auf der die Kritische-Pfad-Analyse beruht (vgl. Abb. 3.5), in Form eines Tracefiles ausgeben zu lassen. Dieses wiederum kann mit Hilfe eines einfachen Konversionsprogramms in ein Format umgesetzt werden, das das für die Animation paralleler Anwendungen entwickelte Werkzeug ParaGraph [HE91] einlesen kann. Somit ist es auch möglich, den idealisierten Simulationsverlauf auf verschiedene Arten graphisch darzustellen. Als Beispiel zeigt Abb. 5.5 sogenannte Raum-Zeit-Diagramme, bei denen für jeden Prozessor die Ausführung von Ereignissen als horizontale Intervalle markiert sind und zwischen erzeugendem und erzeugten Ereignis eine vertikale oder diagonale Linie verläuft, sofern diese auf verschiedenen Prozessoren ausgeführt werden. Bei Abb. 5.5 a) erkennt man sofort, daß es sich um ein weitgehend "inhärent sequentielles" Simulationsmodell handelt: Prozessor 1 ist fast immer mit der Ausführung von Ereignissen beschäftigt, während auf anderen Prozessoren nur gelegentlich ein Ereignis ausgeführt wird, was bei dem analysierten Modell an der ungleichmäßigen Lastverteilung lag. Im Gegensatz dazu bringt das zu Abb. 5.5 b) gehörige Modell gute Voraussetzungen für die parallele Simulation mit, da die kausalen Abängigkeiten zwischen den Ereignissen sich hier offenbar nicht störend auf den parallelen Simulationslauf auswirken.

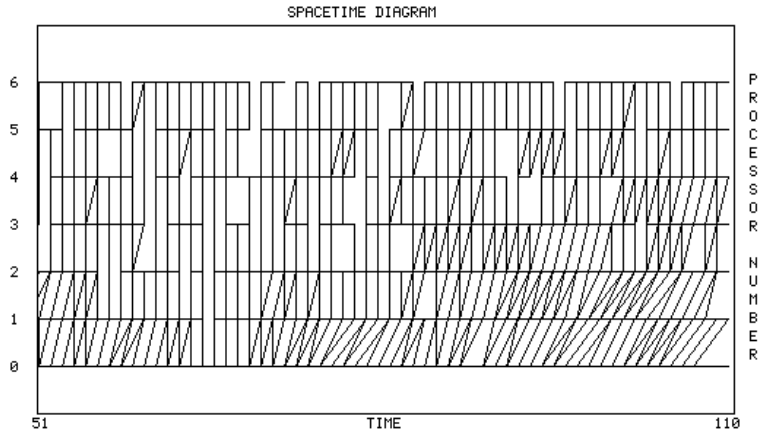
5.3.2 Parallele Simulation mit Orakeldateien

Die Erzeugung von Orakeldateien (vgl. 3.4.2) geschieht im DISQUE-Testbett mit Hilfe des sequentiellen Simulators `simulate`, bei dem durch das Setzen eines Parameters das Protokollieren jedes ausgeführten Ereignis in einer Datei veranlasst werden kann. Um den Zugriff auf diese Daten für den späteren parallelen Simulationslauf so effizient wie möglich zu gestalten, wird die so erzeugte Datei anschließend mit Hilfe eines separaten Programms in mehrere Dateien aufgespalten, die für jeden Rechnerknoten des parallelen Simulators alle dort auszuführenden Ereignisse enthalten. Da die Ereignisse nur die Nummer der Warteschlange, die sie betreffen, enthalten, benötigt das Programm dabei die Zuordnung von Warteschlangen zu Rechnerknoten, die es derselben Datei entnimmt, die auch `cpasim` oder der parallele Simulator benutzen. Die so aufgespaltenen Orakeldaten werden vor Start des parallelen Simulationslaufs auf das parallele Dateisystem des iPSC860 geladen und dort von den Simulatorknoten gelesen.

Die Aufspaltung der Orakeldateien in je eine Datei pro Rechnerknoten erwies sich als großer Effizienzgewinn für die parallele Simulation, da nun die einzelnen Simulatoren nicht mehr fremde Ereignisse "überblättern" mußten. Der zusätzliche Aufwand für den Dateizugriff reduzierte sich dadurch teilweise auf 25% des ursprünglichen Werts und lag dann im Schnitt bei ca. 10% der Gesamtlaufzeit der Simulation. Genauer wird darauf noch in Kapitel 6 eingegangen.



a) Inhärent sequentielle Simulation



b) Echt parallele Simulation

Abbildung 5.5: Animation der Kritischen-Pfad-Analyse mit ParaGraph

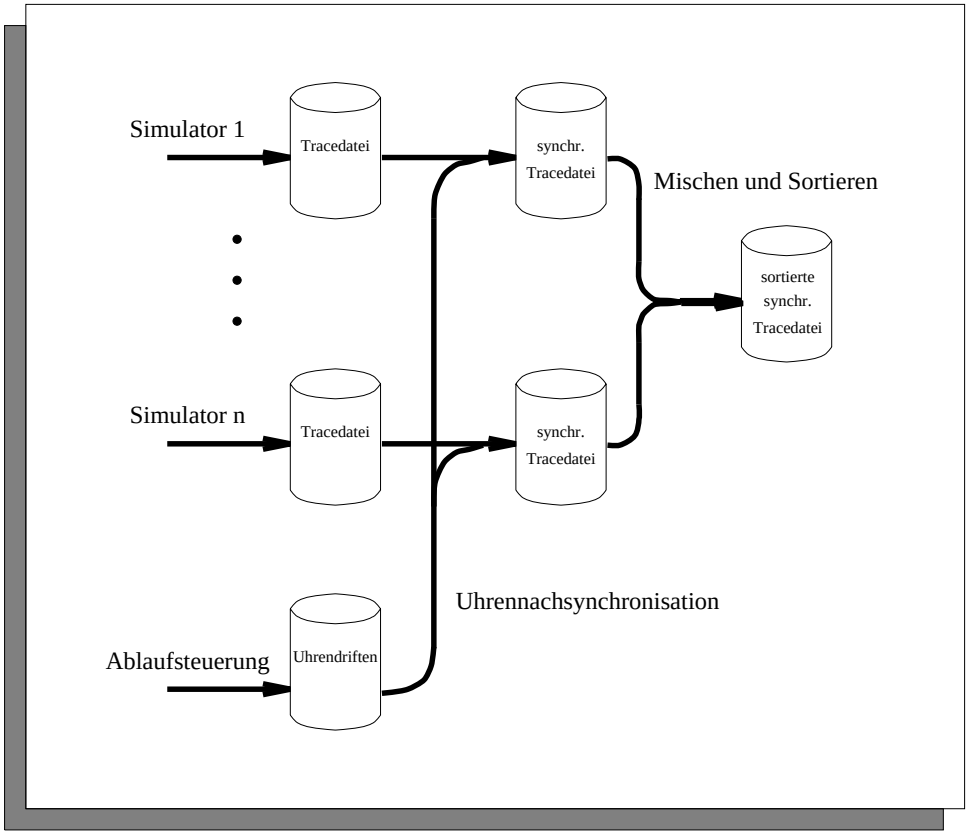


Abbildung 5.6: Aufbereitung der Tracedaten in DISQUE

5.3.3 Leistungsanalyse mit Tracedateien

Das Gewinnen von Daten für die Leistungsanalyse des parallelen Simulators (z. B. Nachrichtenlaufzeiten oder Ausführungsdauer von Ereignissen) wird im DISQUE-Testbett mit Hilfe von Tracedateien in zwei Phasen durchgeführt:

Bei der *Tracedatengewinnung* schreibt jeder Knoten eine Tracedatei auf das parallele Dateisystem des iPSC860. Diese Dateien werden dann zu einer einzigen vereinigt, die die Grundlage für die *Tracedatenauswertung* liefert, bei der die gesuchten Leistungsdaten berechnet werden. Auf diese beiden Phasen wird im folgenden näher eingegangen.

Tracedatengewinnung

Das Erzeugen von Tracedateien (Abb. 5.6) im DISQUE-Testbett geschieht mit einer dafür instrumentierten Version des verteilten Simulators. Die einzelnen Rechnerknoten schreiben ihre Traceeinträge auf getrennte Dateien des parallelen Dateisystems, um den Ablauf der Simulation nicht zusätzlich durch das Synchronisieren des Dateizugriffs zu beeinträchtigen. Protokolliert werden dabei folgende Vorgänge:

- Start/Ende einer Ereignisausführung
- Senden/Empfangen einer Nachricht
- Erkennen eines Deadlock
- Erzeugen/Löschen eines Ereignisses
- Erreichen des Simulationsendes

Das genaue Format der Traceeinträge findet man in [Stu93]. Durch die Instrumentierung verlängert sich dabei die Laufzeit des parallelen Simulators um ca. 35%, wovon ca. 15% auf die Aufbereitung der Tracedaten (Zeitmessungen, etc.) entfielen.

Für die Erzeugung der Komponenten der Traceeinträge, die Realzeitstempel darstellen (z. B. Start- und Endzeitpunkt einer Ereignisausführung), stehen auf den Rechnerknoten des iPSC860 Hardware-Uhren mit einer Auflösung von 100 ns und einer Zeitstempellänge von 64 bit zur Verfügung, so daß Zeitstempelüberschläge auf dieser Maschine kein Problem darstellen. Leider sind diese Uhren nicht synchronisiert, weswegen diese Aufgabe von der Ablaufsteuerung mitübernommen wird. Die Methode, mit der dies geschieht, stammt aus dem NTP-Protokoll [Mil89], das für die Uhrensynchronisation von Rechnern auf dem Internet verwendet wird. Die Ablaufsteuerung ermittelt dazu die Differenz zwischen ihrer Uhr und der Uhr eines Simulatorknotens, indem sie eine Nachricht an den Knoten schickt, die dieser wiederum zurückschickt. Die Ablaufsteuerung speichert dabei die Zeiten t_1 bei Absenden der Nachricht und t_2 beim Empfang der Rückantwort. Der Simulatorknoten schickt mit der Rückantwort die Zeit t_3 (laut seiner Uhr) beim Empfang der Nachricht von der Ablaufsteuerung mit (Abb. 5.7). Die Uhrendifferenz errechnet sich dann als $\frac{t_2-t_1}{2} - t_3$. Dieses Verfahren liefert natürlich nur dann exakte Werte, wenn die Nachrichtenlaufzeiten für beide Nachrichten identisch sind und es zwischen Ankunft der Nachricht und Absenden der Antwort auf dem Simulatorknoten nicht zu Verzögerungen kommt. In der Praxis ist dies so gut wie nie erfüllt. Praktische Erfahrung bei der Entwicklung des NTP-Protokolls zeigte jedoch, daß im Regelfall die Uhrendifferenz um so genauer approximiert wurde, je kleiner die Umlaufzeit $t_2 - t_1$

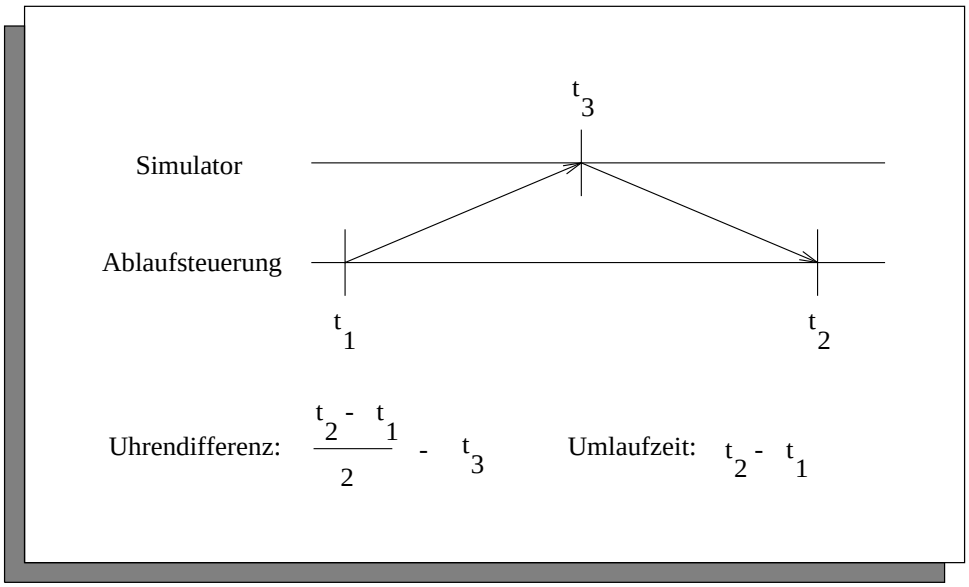


Abbildung 5.7: Berechnen der Differenz zweier Uhren

der Nachricht war. Daher wird diese Messung mehrfach wiederholt und nur ein gewisser Teil der Meßwerte mit den kleinsten Umlaufzeiten für die Mittelwertbildung über die Uhrendifferenzen benutzt. (Die Ablaufsteuerung des parallelen Simulators benutzt von 100 Messungen die 10 mit den geringsten Umlaufzeiten, was sich als völlig ausreichend erwies.) Dieser Mittelwert wird dann an den Simulator geschickt, worauf dieser seine Uhr entsprechend korrigiert.

Ein weiteres Problem im Zusammenhang mit Hardware-Uhren ist ihre unterschiedliche Geschwindigkeit. Konkrete Messungen am iPSC860 ergaben, daß sich die Uhrendifferenzen zwischen zwei Rechnerknoten um bis zu $20\mu s$ pro Sekunde verändern kann, was schon nach wenigen Minuten Laufzeit zu Anomalien in den Tracedateien führt (z. B. in Form von Nachrichten, die ankommen, bevor sie abgeschickt wurden). Um dies zu vermeiden, führt die Ablaufsteuerung nach Ende der Simulation dieselbe Uhrensynchronisation wie vor dem Start noch einmal durch und speichert für jeden Simulatorknoten die Differenz der beiden Werte in einer Datei. Diese bildet dann die Grundlage für ein Programm, das die Realzeitinformationen der Traceeinträge aller Tracedateien noch einmal nachsynchronisiert, wobei es davon ausgeht, daß sich die Uhrendifferenz zwischen zwei Rechnerknoten linear ändert. Diese Annahme erwies sich zwar für den iPSC860 als richtig, für das SUN-Workstationnetz jedoch als nicht haltbar: Abb. 5.8 zeigt die Uhrendifferenzen zwischen zwei SUN-Workstations vom Typ ELC und zwischen zwei Knoten des iPSC860, die mit der oben beschriebenen Methode wiederholt über einen längeren Zeitraum gemessen wurden. Wie man sieht, ändert sich die Uhrendifferenz zwischen den beiden SUN-Workstations

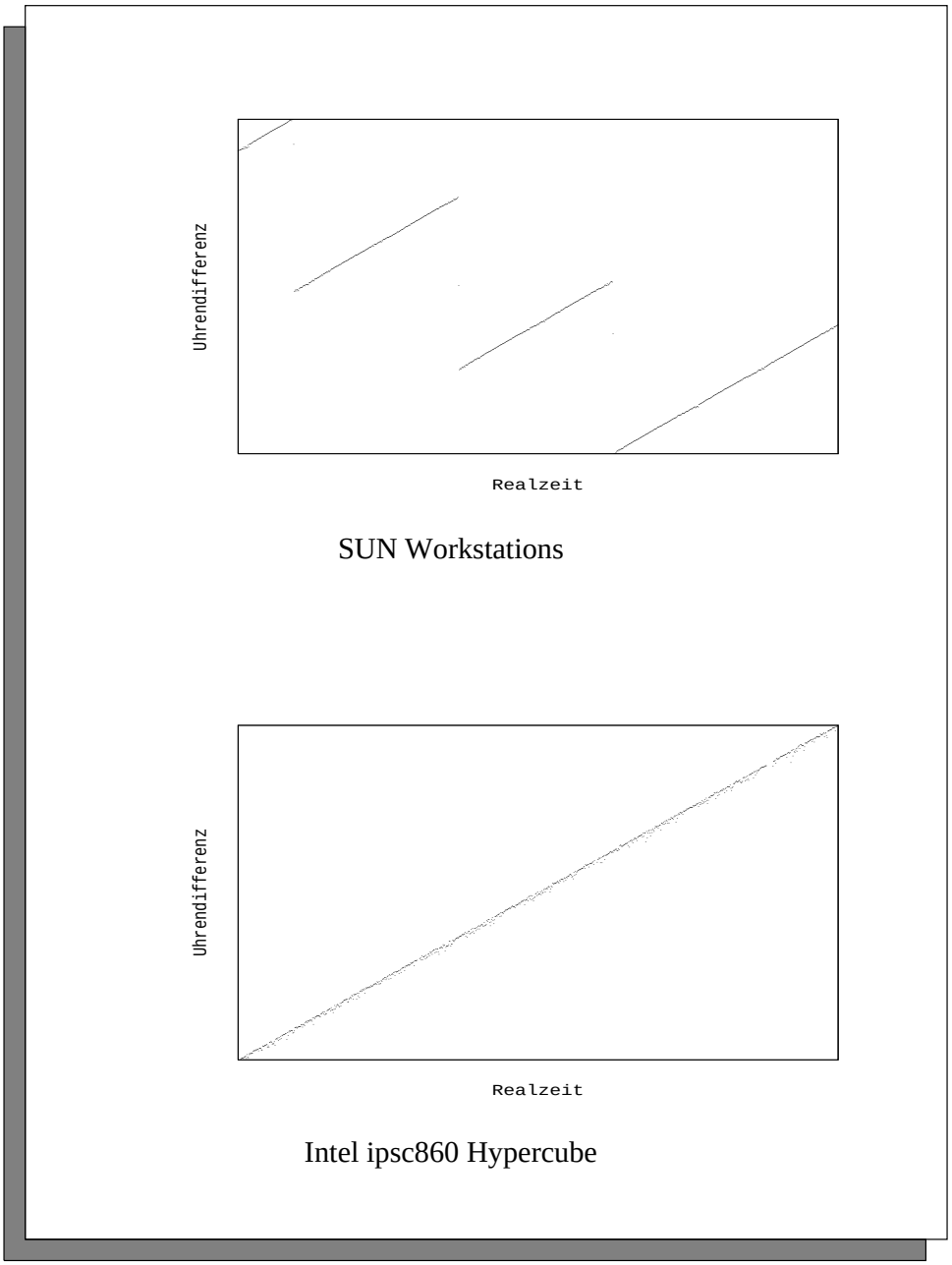


Abbildung 5.8: Veränderung der Uhrendifferenzen bei verschiedenen Rechnern

zwar linear, enthält aber gewisse Sprünge, die wohl auf eine Uhrensynchronisation durch die UNIX-Kerne der Rechner zu unterschiedlichen Zeiten zurückzuführen sind. Beim iPSC860 dagegen entfallen diese Sprünge. Entsprechend wiesen die mit dem iPSC860 erzeugten Tracedateien nach der Nachsynchronisation keinerlei Anomalien mehr auf, während die Nachsynchronisation der mit dem SUN-Workstationnetz erzeugten Tracedateien ohne den genauen zeitlichen Verlauf der Uhrendifferenzen nicht möglich war.

Zum Abschluß der Tracedatengewinnung werden sämtliche Tracedateien zu einer Gesamtdatei zusammengemischt und nach dem Realzeitstempel ihrer Erzeugung sortiert, um den Werkzeugen der Tracedatenauswertung die Vorgänge im Simulator in chronologischer Reihenfolge zu präsentieren.

Tracedatenauswertung

Die Werkzeuge zur Auswertung von Tracedateien basieren auf dem an der Universität Erlangen entwickelten POET-System [Moh90]. In ihm werden die Formate der verschiedenen Traceeinträge einer Tracedatei in einer für dieses System entwickelten Sprache (TDL) beschrieben, wobei die einzelnen Felder eines Traceeintrags symbolische Namen erhalten. Durch eine vom POET-System zur Verfügung gestellte Bibliothek ist es dann möglich, Werkzeuge zu entwickeln, die ausschließlich über diese symbolischen Namen auf die Inhalte der einzelnen Felder eines Traceeintrags zugreifen. Die Bibliotheksfunktionen benötigen dazu lediglich eine speziell aufbereitete Version der TDL-Beschreibung der Tracedatei. Dies verbessert die Wartbarkeit des Codes der Werkzeuge, da Änderungen am Format der Tracedateien auf diese Weise häufig nur Änderungen an der TDL-Beschreibung, nicht aber an den Werkzeugen selber nach sich ziehen. Weiterhin gibt es im POET-System eine Reihe von Werkzeugen, wie etwa `list` zur formatierten Anzeige und `checktrace` zum Überprüfen der Konsistenz von Tracedateien, die mit Hilfe der entsprechenden TDL-Beschreibung auf beliebigen Tracedateien arbeiten können, was sich insbesondere als große Hilfe bei der Entwicklung der Instrumentierung des parallelen Simulators erwies.

Aufbauend auf POET wurde für das DISQUE-Testbett das Werkzeug YES entwickelt, das die mit dem im vorangegangenen Abschnitt beschriebenen Verfahren gewonnenen Tracedateien auswertet. Im wesentlichen lassen sich mit ihm folgende Daten ermitteln:

- Gesamtzahl der ausgeführten/gelöschten/neu erzeugten Ereignisse
- Gesamtzahl der versendeten Nachrichten eines bestimmten Typs (Ereignis-, Nullnachricht, etc.)

- Dauer und statistische Verteilung der Ausführung von Ereignissen/Idlezeiten der Simulatorknoten/Nachrichtenlaufzeiten
- zeitlicher Verlauf von tatsächlicher und approximierter GVT
- Länge von Ereignislisten

Die Menuesteuerung von YES wurde graphisch mit MOTIF unter XWindows realisiert, die Darstellung der Ergebnisse erfolgt im Augenblick noch in Form von Tabellen, jedoch wird an einer graphischen Ergebnisausgabe gearbeitet [Mar94].

Da die Tracedateien nur die notwendigsten Daten zur Rekonstruktion des parallelen Simulationslaufs enthalten, werden sie vor der Auswertung mit YES von einem Präprozessor überarbeitet. Dieser ergänzt einerseits vorhandene Traceeinträge um weitere Felder (z. B. beim Erkennen eines Deadlock um die Zeitdauer zwischen Auftreten und Erkennen eines Deadlock), fügt aber andererseits auch neue Einträge hinzu (z. B. für den Anfang der Idlezeit eines Prozessors nach der Ausführung des letzten Ereignisses seiner Ereignisliste). Der Präprozessor wurde ebenfalls aufbauend auf POET entwickelt, seine Funktionsweise wird in [Stu93] beschrieben.

Erfahrungen mit tracebasierter Leistungsanalyse

Beim praktischen Einsatz von YES ergaben sich leider eine Reihe von Problemen: Die Aufbereitung der Tracedateien (Uhrensynchronisation, Mischen und Sortieren, Präprozessor) erwies sich als zeitraubend, insbesondere da für realistische Simulationsläufe Tracedateien in der Größe mehrerer Megabytes verarbeitet werden mußten. Hierbei zeigte sich ein generelles Problem mit der Leistungsanalyse anhand von Tracedateien: Vielfach sind für ein konkretes Problem (z. B. das vom Simulationslauf verursachte Nachrichtenaufkommen) nur wenige Einträge einer Tracedatei interessant, verarbeitet werden muß aber immer die gesamte Tracedatei. Daher ist es vielfach günstiger, "einfache" Werte wie etwa die Anzahl gesendeter Ereignisnachrichten während des Simulationslaufs direkt zu berechnen. Leider gibt es jedoch Werte, deren Berechnung eine globale Sicht auf das System erfordern und die daher erst post mortem anhand einer Tracedatei berechnet werden können (etwa die Zeit, die zwischen Auftreten und Erkennen eines Deadlocks vergeht).

Weiterhin stellte sich heraus, daß der Einsatz von Multithreading auf den Rechnerknoten spezielle Anforderungen an das Format der Tracefiles stellt. Ein gutes Beispiel dafür ist die Dauer der Ausführung von Ereignissen: In dem von YES verwendeten Tracedateiformat gibt es für Start und Ende der Ereignisausführung

jeweils einen Eintrag. Die Differenz der Realzeitstempel der beiden Einträge ergibt dann gerade die Dauer der Ausführung des Ereignisses. Hierbei wird leider nicht berücksichtigt, daß der ausführende Prozeß während der Ereignisausführung deaktiviert und ein anderer Prozeß aktiviert werden kann, so daß sich das Ende der Ereignisausführung entsprechend verzögert. Folglich müßte die Dauer der Ereignisausführung schon während des Simulationslaufs mit Hilfe eines pro Prozeß geführten Timers (ähnlich der `clock` Funktion unter UNIX) berechnet und im Traceeintrag für das Ausführungsende vermerkt werden.

Insgesamt erwies sich damit die Leistungsanalyse mit YES als zu umständlich und zu unpräzise, um im Rahmen dieser Arbeit zur Ergebnisanalyse eingesetzt zu werden. Statt dessen wurden die benötigten Werte mit Simulationsläufen, bei denen die Simulatoren entsprechend instrumentiert waren, direkt gemessen.

Kapitel 6

Messungen und Analysen mit DISQUE

Wie aus der Literatur bekannte, praktische Erfahrungen mit paralleler, ereignisgesteuerter Simulation gezeigt haben, ist es im Regelfall nicht möglich, annähernd lineare Beschleunigungen zu erzielen; vielfach liegen sogar die Beschleunigungswerte um den Wert 1 oder darunter. Denkbare Gründe dafür gibt es viele:

- Das Simulationsmodell muß an die besonderen Anforderungen des parallelen Simulators (Partitionierung des Zustandsraums) angepaßt werden, was die Simulation des Modells von vornherein aufwendiger macht.
- Die Last auf den einzelnen Simulatorknoten ist ungleichmäßig verteilt.
- Die kausalen Abhängigkeiten zwischen den Ereignissen behindern ihre parallele Ausführung.
- Der parallele Simulator muß eine Reihe von Arbeiten zusätzlich zur eigentlichen Simulation durchführen (Lesen und Schreiben von Nachrichten, Semaphor-Operationen etc.).
- Das (konservative oder optimistische) Synchronisationsverfahren verzögert die Ausführung von Ereignissen und/oder erzeugt zusätzlichen Aufwand.

Was dabei bisher kaum untersucht wurde, ist, wie stark diese (und andere) Faktoren in der Praxis die mit paralleler Simulation erzielbare Beschleunigung grundsätzlich beschränken. Dies ist Gegenstand der in diesem Kapitel beschriebenen Messungen

und Analysen. Der grundlegende Gedanke dabei ist, die Laufzeit jeweils zweier unterschiedlicher Simulatoren zu vergleichen, die sich nur in einem der eben geschilderten Punkte unterscheiden. Diese Vergleiche werden mit einem einheitlichen Satz von Simulationsmodellen durchgeführt, deren Beschreibung sich in 6.1 findet.

In 6.2 werden die Laufzeiten der beiden sequentiellen Simulatoren `optsim` und `simulate` miteinander verglichen. Wie in 5.2.1 bereits erwähnt, zeigt dieser Vergleich, wie stark sich durch die notwendige "Parallelisierung" des Simulationsmodells der Aufwand für dessen Simulation erhöht.

In 6.3 geht es um Analysen mit `cpasim`. Dort werden zunächst die Ergebnisse der Analyse der Lastbalancierung untersucht. Diese stellt letztendlich das Ergebnis des Vergleichs eines sequentiellen Simulationslaufs mit einem (hypothetischen) parallelen Simulationslauf dar, bei dem allen Simulatoren von vornherein bekannt ist, welche Ereignisse sie auszuführen haben (vgl. 3.4.1), so daß vom Time-Advancement-Problem und seiner oft aufwendigen Lösung mit konservativen oder optimistischen Synchronisationsverfahren abstrahiert wird. Der Vergleich zeigt dann, wie stark sich die unterschiedliche Verteilung der Lasten auf den einzelnen Simulatorknoten auf die Laufzeit der parallelen Simulation auswirkt. Anschließend wird das Ergebnis der Kritischen-Pfad-Analyse betrachtet. Hierbei wird wiederum ein hypothetischer paralleler Simulationslauf, bei dem im Gegensatz zu dem gerade eben geschilderten Ereignisse erst ausgeführt werden können, nachdem sie erzeugt worden sind (vgl. wiederum 3.4.1), mit der Laufzeit eines sequentiellen Simulators verglichen. Das Ergebnis zeigt dann, wie stark sich ungünstige Lastbalancierung und kausale Abängigkeiten auf die Laufzeit des parallelen Simulators auswirken. Die Auswirkungen der kausalen Abhängigkeiten allein erhält man durch den Vergleich der Werte der Analyse der Lastbalancierung und der Kritischen-Pfad-Analyse, da dies einen Vergleich der Laufzeiten der beiden hypothetischen Simulationsläufe (beide dividiert durch die Laufzeit einer sequentiellen Simulation) darstellt.

In 6.4 wird untersucht, wie sich der zusätzlich zur eigentlichen Simulation anfallende Aufwand bei einem parallelen Simulator auswirkt. Zu diesem Zweck vergleicht man die Laufzeit des sequentiellen Simulators `simulate` mit der eines parallelen Simulators, bei dem nur ein einziger Simulatorknoten zur Verfügung steht. Da bei letzterem alle Teilmodelle zu einem einzigen geclustert sind, handelt es sich dabei um einen sequentiellen Simulator, der dasselbe Modell wie `simulate` simuliert, bei dem aber zusätzlich noch die bei einem parallelen Simulator notwendigerweise anfallenden Operationen (wie z. B. Zugriffe auf Semaphore zur Synchronisation) durchgeführt werden.

In 6.5 werden schließlich die Auswirkungen der durch konservative Synchronisationsverfahren verursachten Verzögerung der Ereignisausführung untersucht. Hierzu

vergleicht man die Laufzeiten eines parallelen Simulators, der mit Orakeldateien fast ideal synchronisiert wird (vgl. 5.3.2), mit der eines nach der Linktime-Methode synchronisierten, ebenfalls parallelen Simulators. Weiterhin werden auf dieselbe Art konservative Nullnachrichten-Verfahren untersucht.

6.1 Untersuchte Modelle

Die Suche nach Warteschlangennetzen, die für praktische Anwendungen entwickelt wurden, erwies sich als etwas problematisch. Dies lag in erster Linie an der fehlenden Existenz frei verfügbarer Bibliotheken, aber auch daran, daß die in DISQUE verwendete Beschreibungssprache LAVENDER nirgendwo sonst benutzt wird. Ein Großteil der im Rahmen dieser Arbeit verwendeten Modelle wurde daher künstlich für die Experimente erzeugt. Dies geschah stets auf dieselbe Weise: Zunächst wurde die Topologie der Simulatorknoten (also die Topologie, die das geclusterte Simulationsmodell aufweisen sollte) festgelegt. Danach wurde eine bestimmte Anzahl von Warteschlangen auf die Rechnerknoten verteilt und dort hintereinander aufgereiht, wobei in die erste Warteschlange alle Eingänge des Simulatorknotens hinein- und aus der letzten Warteschlange alle Ausgänge des Simulatorknotens herausführen (vgl. z. B. Abb. 6.2). Die Warteschlangen wurden dabei so auf die Simulatorknoten verteilt, daß eine möglichst gleichmäßige Verteilung der Simulationslast entstand. Auf diese Weise wurde das Problem der Lastbalancierung bei der parallelen Simulation umgangen. Weiterhin war es möglich, durch die Variation der Zahl der Warteschlangen auf den Simulatorknoten Serien von Modellen zu erzeugen, bei denen die geclusterten Versionen alle dieselbe Topologie aufwiesen.

Die Erzeugung der LAVENDER-Beschreibungen und Partitionierungsdateien der Modelle geschah mit Hilfe von C++-Programmen. Die Bedienzeitverteilungen der Warteschlangen wurden einheitlich negativ exponential-verteilt mit dem Mittelwert 0.9 und die Zwischenankunftzeit der Quellen konstant 1.0 gewählt, so daß nur selten Ankunfts- und Abgangsereignisse mit identischen Zeitstempeln vorkamen. Auf diese Weise wurden Anomalien beim Zugriff auf gewisse Datenstrukturen wie etwa die als Ereignislisten verwendeten Calendar-Queues vermieden. Wegen der umfangreichen Messungen und der beschränkten Verfügbarkeit höherer Knotenzahlen auf dem iPSC860 wurden alle Modelle für die Simulation mit sieben Simulator- und einem Ablaufsteuerungsknoten konstruiert.

Abb. 6.1 zeigt die Größe der einzelnen Warteschlangennetze der Modellreihen. Für sie gilt im einzelnen folgendes: Bei den Modellen der `tandem`-Reihe (Abb. 6.2) handelt es sich um eine Kette von hintereinandergeschalteten Warteschlangen, an deren

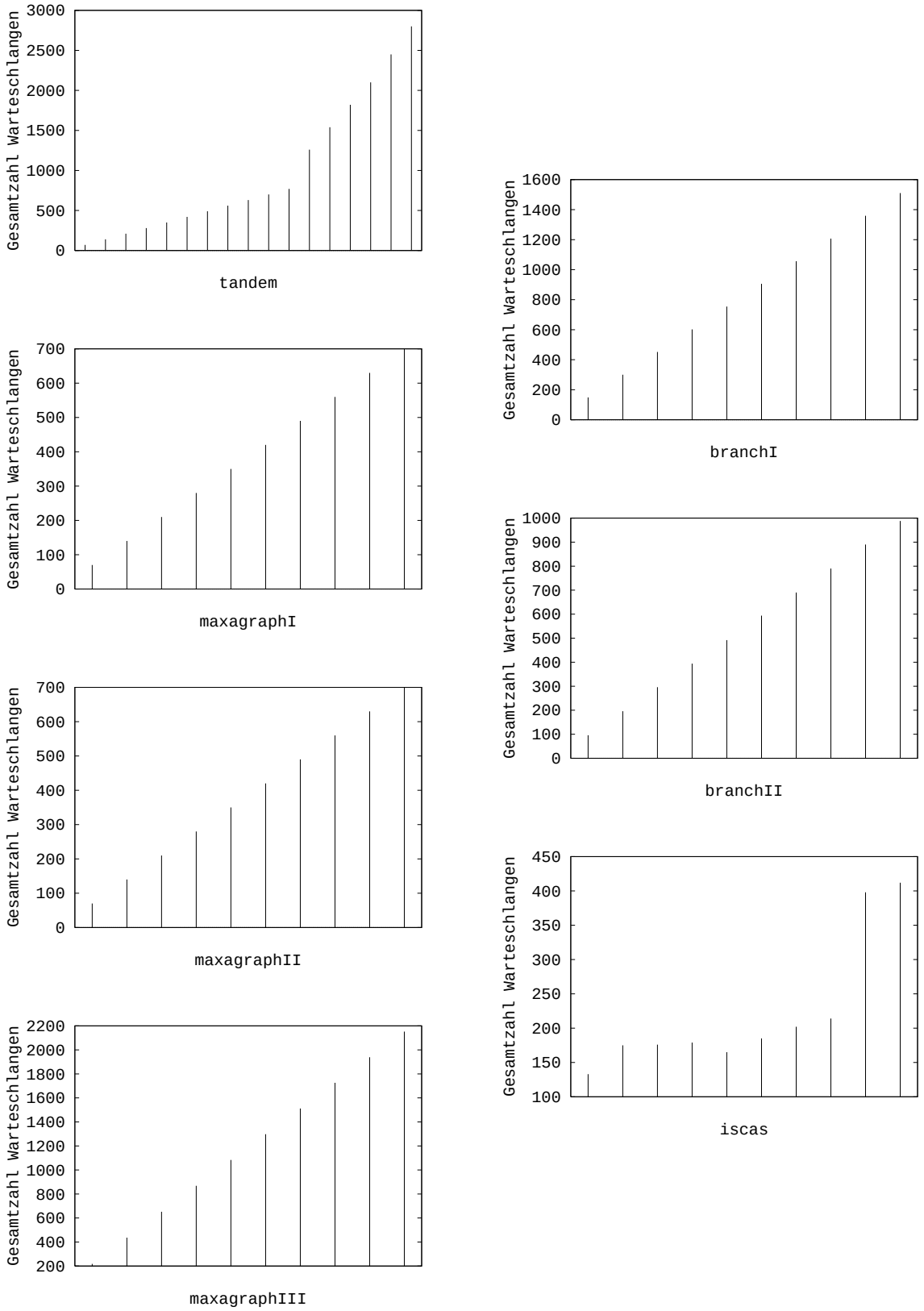


Abbildung 6.1: Größe der Warteschlangennetze

Anfang sich eine Kundenquelle und an deren Ende sich eine Kundensenke befindet. Die Warteschlangen wurden dabei gleichmäßig auf die Simulatorknoten verteilt. Um die Phase des Füllens der Warteschlangen mit Kunden (und die daraus resultierende ungleiche Verteilung der Simulationslast am Anfang der Simulation) zu umgehen, enthält jede Warteschlange zu Beginn der Simulation zwei Kunden. Wie in 6.3 noch genauer gezeigt wird, können die Ereignisse von den Simulatorknoten fast "ungebremst" parallel ausgeführt werden, was mit dem geringen Verzweigungsgrad der Topologie der Simulatorknoten zu tun hat. Daher wurde anhand dieser Modelle untersucht, welche Ergebnisse sich mit paralleler, ereignisgesteuerter Simulation erzielen lassen, wenn das Modell dazu nahezu ideale Voraussetzungen liefert.

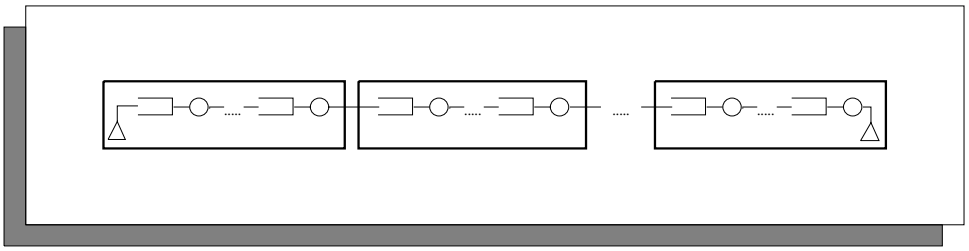


Abbildung 6.2: Struktur der `tandem`-Warteschlangennetze

Abb. 6.3 zeigt die Toplogie der Simulatorknoten bei den `maxagraph`-Modellen. Es handelt sich dabei um einen Graphen mit $\binom{7}{2}$ Kanten, also um einen azyklischen gerichteten Graphen mit maximaler Kantenzahl. Daher eignen sich diese Modelle besonders gut, um den Einfluß der aus den Verzweigungen in der Topologie der Simulatorknoten herrührenden kausalen Abhängigkeiten zwischen den Ereignissen verschiedener Knoten auf das Ergebnis der Kritischen-Pfad-Analyse und auf die Laufzeiten paralleler Simulationsläufe mit konservativen Synchronisationsverfahren zu untersuchen. Die Verzweigungswahrscheinlichkeiten für Kunden, die die letzte Warteschlange eines Simulatorknotens verlassen, sind für alle Verzweigungen aus dem Knoten gleich (d. h. $\frac{1}{7-i}$ für alle von Knoten i ausgehenden Verzweigungen in Abb. 6.3). Die hieraus resultierenden Probleme mit ungleicher Simulationslast (die darauf beruht, daß die Simulatorknoten unterschiedlich häufig Kunden von anderen Knoten erhalten) wurden bei den Modellreihen `maxagraphI`, `maxagraphII`, `maxagraphIII` unterschiedlich gelöst:

Bei `maxagraphI` wurde die erste Warteschlange auf jedem Simulatorknoten mit einer Kundenquelle versehen, die für die Warteschlangen des Knotens zusätzliche Kunden erzeugt. Bei `maxagraphII` wurde diese Quelle nur für Knoten 1 beibehalten und statt dessen alle Warteschlangen mit 2 Kunden bei Start der Simulation belegt, um so die Probleme des Auffüllens des Netzes mit Kunden zu vermeiden. Bei `maxagraphIII` schließlich wurden statt der beim Start vorhandenen Kunden die Anzahl der Warteschlangen auf den Simulatorknoten entsprechend variiert, so daß bei

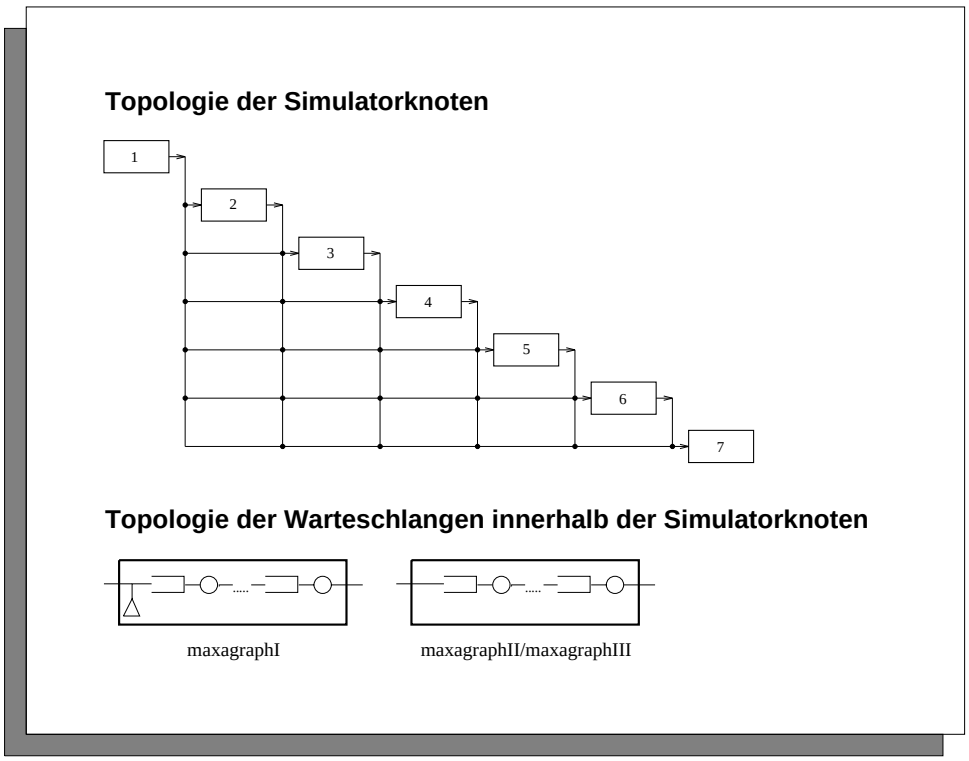


Abbildung 6.3: Struktur der `maxagraph`-Warteschlangennetze

Knoten mit geringerer Anzahl von Kunden die Simulationslast erhöht ist, da die Anzahl der Warteschlangen, die ein Kunde dort passieren muß, vergrößert ist. Da Knoten i ($2 \leq i \leq 7$) im Schnitt von $\frac{1}{8-i}$ der Kunden, die Knoten 1 aus seiner Kundenquelle erhält, erreicht wird, wurde die Anzahl seiner Warteschlangen auf das $(8-i)$-fache der Anzahl der Warteschlangen von Knoten 1 erhöht. Die Anzahl der Warteschlange von Knoten 1 wurde bei allen `maxagraph`-Modellen von 10 ausgehend in 10er-Schritten bis 100 erhöht und bei den anderen Knoten entsprechend angepaßt. Bei den `maxagraphIII`-Modellen wurde anschließend die Zahl der Warteschlangen auf Knoten 1 um einen experimentell ermittelten Faktor reduziert, um so die durch die Quelle dort entstandene zusätzliche Last auszugleichen.

Die `branch`-Modelle wurden konstruiert, um die Auswirkungen von Kanten in der Topologie der Simulatorknoten zu untersuchen, entlang denen nur selten Ereignisse verschickt werden, was insbesondere bei der parallelen Simulation mit konservativen Synchronisationsverfahren eine Rolle spielt. Abb. 6.4 zeigt die Topologie der Simulatorknoten mit den jeweiligen Verzweigungswahrscheinlichkeiten. Bei den `branchI`-Modellen wurde für die vier Verzweigungen (d. h. die Kanten von Knoten 1 zu den Knoten 2, 3, 4 und 5) 0.6, 0.6^2, 0.6^3 und $1-(0.6+0.6^2+0.6^3)$ gewählt. Die einzige Kundenquelle befindet sich dort bei der ersten Warteschlange von Knoten 1. Die

daraus resultierende ungleiche Verteilung der Simulationslast auf den Knoten 2 bis 5 wurde wie bei den `maxagraphIII`-Modellen durch eine entsprechend erhöhte Anzahl von Warteschlangen auf den Knoten ausgeglichen, die experimentell ermittelt wurde. Die folgende Tabelle zeigt die Werte für das kleinste Modell. Bei allen anderen Modellen wurden die Zahlen für alle Knoten ver-n-facht ($1 \leq n \leq 10$).

Knoten-Nr.	Anzahl Warteschlangen
1	10
2	15
3	23
4	42
5	50
6	10

Entsprechend wurde bei den `branchII`-Modellen vorgegangen (Abb. 6.4). Die Idee bei diesen Modellen war, durch das Hintereinandersetzen von zwei Verzweigungen zu untersuchen, wie sehr die Verzögerung der ersten Verzweigung diejenige der zweiten Verzweigung verstärkt. Auch hier wurde durch Anpassen der Anzahl der Warteschlangen für gleichmäßige Simulationslast auf den Rechnerknoten gesorgt und größere Modelle durch Vervielfachen der Warteschlangenanzahlen erzeugt. Die folgende Tabelle zeigt wiederum die Anzahlen für das kleinste Modell.

Knoten-Nr.	Anzahl Warteschlangen
1	10
2	12
3	22
4	10
5	11
6	22
7	10

Bei den `iscas`-Modellen handelt es sich um aus VLSI-Schaltungen abgeleitete Warteschlangennetze. Dazu wurden bei einigen Schaltungen aus der ISCAS-Benchmark-Suite [BBK89] Gatter durch Warteschlangen und Eingänge für Stimuli durch Quellen für die entsprechenden Warteschlangen ersetzt. Es ging dabei nicht darum, das Verhalten der Schaltung durch das entsprechende Warteschlangennetz zu modellieren, sondern Warteschlangennetze mit unregelmäßigen, aber realistischen Topologien zu untersuchen. Die Verteilung der Warteschlangen auf die Simulatorknoten wurde mit Hilfe des in 5.2.2 bereits erwähnten azyklischen Partitionierungsverfahren aus [SS90] durchgeführt (vgl. Abb. 6.5). Dabei wird zunächst das Warteschlangennetz als gerichteter Graph aufgefaßt und in diesem die starken Zusammenhangskomponenten bestimmt. Diese werden bezüglich der Halbordnung, die durch die zwischen

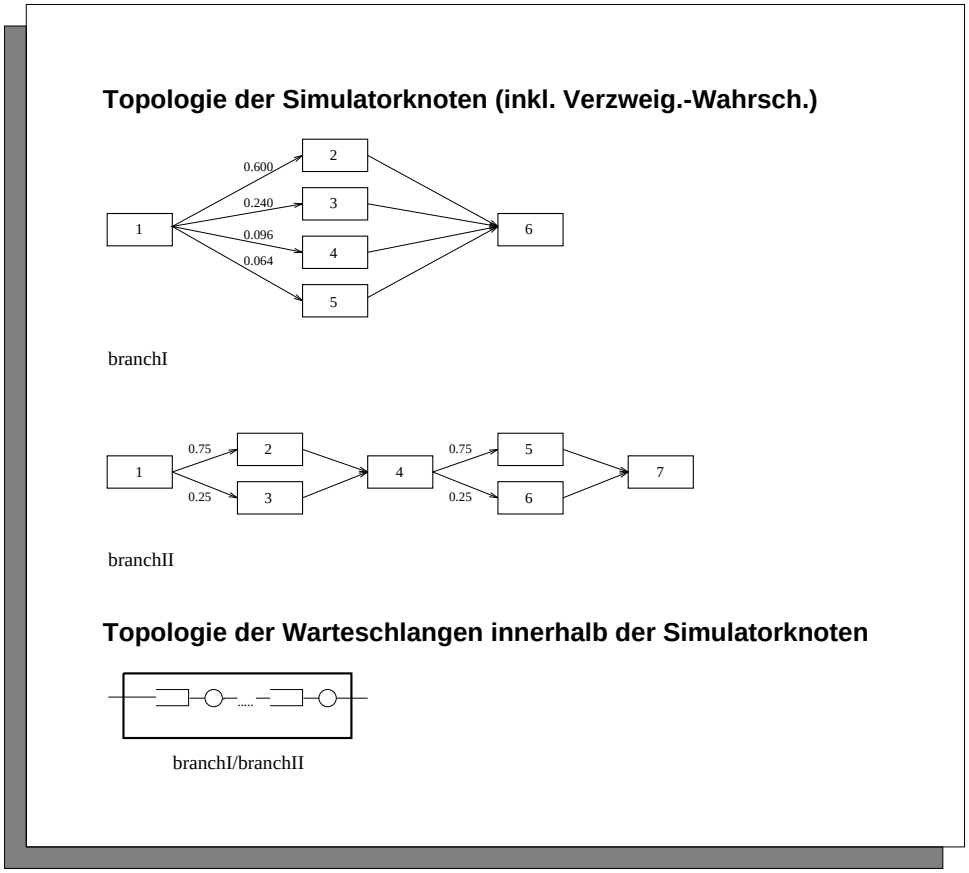


Abbildung 6.4: Struktur der `branch`-Warteschlangennetze

den Zusammenhangskomponenten verlaufenden Kanten gegeben ist, topologisch sortiert und in der so ermittelten Reihenfolge auf die Simulatorknoten verteilt. Auf diese Weise entsteht eine zyklenfreie Simulatorknoten-Topologie, so daß trotz des fehlenden Lookaheads (vgl. 3.2) die Modelle mit konservativen Verfahren parallel simulierbar sind. Bei der Verteilung der starken Zusammenhangskomponenten versucht das Verfahren, eine möglichst gleichmäßige Verteilung der Simulationslast zu erreichen, indem es vor der Zuweisung einer Zusammenhangskomponente an einen Simulatorknoten den Durchschnitt der noch pro Simulatorknoten zu vergebenden Simulationslast ausrechnet und diesen für den aktuellen Simulatorknoten so genau wie möglich zu erreichen versucht. Die Simulationslast der Zusammenhangskomponenten errechnet der Algorithmus dabei anhand einer Datei, in der mit Hilfe von `simulate` für jede Warteschlange die Summe der Ausführungszeiten ihrer Quellen-, Ankunfts- und Abgangsereignisse protokolliert wurden. Dieses Verfahren funktioniert natürlich nur, falls die Anzahl der starken Zusammenhangskomponenten größer als die Anzahl der vorhandenen Simulatorknoten ist. Bis auf eine einzige Ausnahme

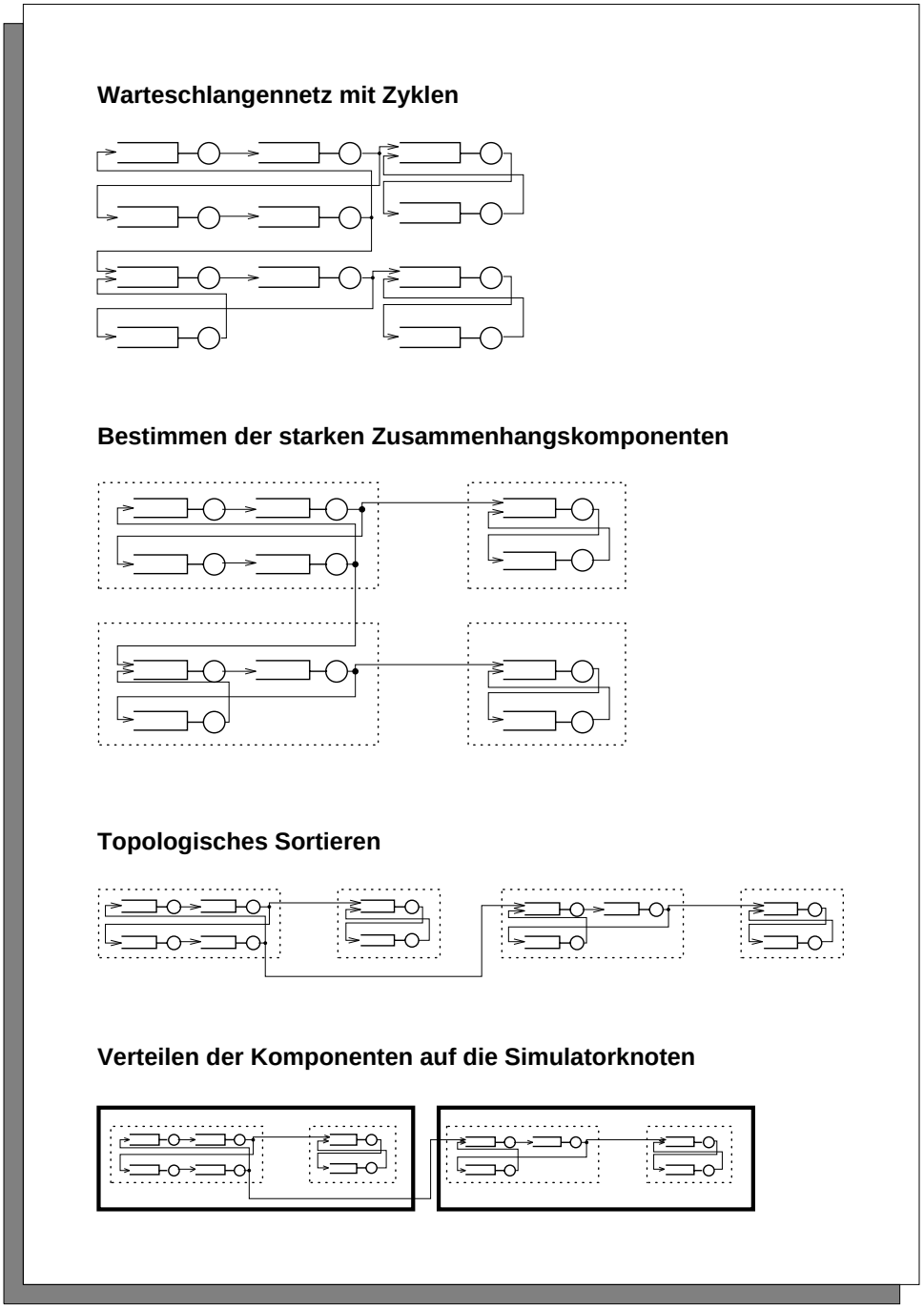


Abbildung 6.5: Azyklisches Partitionieren von Warteschlangennetzen

war dies jedoch für sämtliche aus der ISCAS-Suite abgeleiteten Modelle der Fall.

Einen ersten Einblick in die Probleme bei der parallelen Simulation dieser Modelle liefern die Ausführungszeiten der Ereignisse. Die Mittelwerte und Standardabweichungen dieser Zeiten sind in den Abb. 6.6 - 6.12 dargestellt (als Intervall der Länge der doppelten Standardabweichung um den Mittelwert), gemessen wurden sie mit `cpasim` auf dem iPSC860. Sie zeigen recht deutlich, daß die Ausführungszeiten im Mittel von der Art und Größe des simulierten Warteschlangennetzes weitgehend unabhängig sind und ca. 60 μs für Quellen- sowie ca. 100 μs für Ankunfts- und Abgangsereignisse betragen. (Das unregelmäßige Aussehen der Werte in Abb. 6.12 im Gegensatz zu allen anderen Werten erklärt sich daraus, daß es sich bei den `iscas`-Modellen um eine Sammlung verschiedener Warteschlangennetze handelt und nicht, wie bei den anderen Modellreihen, um Warteschlangennetze mit jeweils stark ähnlicher Topologie.) Diese Zahlen liefern einen Richtwert für die Beurteilung der Dauer von Systemoperationen wie etwa Senden oder Empfangen von Nachrichten, Zugriffe auf die Ereignisliste, Semaphor-Operationen etc., wenn diese bei der Ausführung von Ereignissen regelmäßg durchgeführt werden. Da Quellenereignisse nur vergleichsweise selten vorkommen (vgl. hierzu die in den Abb. 6.6 - 6.12 dargestellten Anzahlen ausgeführter Ereignisse), sind 100 μs damit so etwas wie die "Einheit", in der der von den erwähnten Operationen verursachte zeitliche Aufwand gemessen werden kann. Wie sich später noch zeigen wird, sind viele der bei der parallelen Simulation zusätzlich notwendigen Operationen damit gemessen "teuer".

Abschließend sei noch auf ein Phänomen hingewiesen, das wohl häufig bei C-Compilern vorkommt. Es äußert sich in den überdurchschnittlich hohen Ausführungszeiten für Quellen- und Ankunftsereignisse bei den `maxagraphI`- und `iscas`-Modellen. Nähere Untersuchungen ergaben, daß es sich bei diesen Modellen um Warteschlangennetze ohne Gleichgewichtszustand handelt, da die dort vorhandenen Quellen mehr Kunden produzieren als die Warteschlangen, die von ihnen gespeist werden, verarbeiten können. Daher muß vom Simulator ständig neuer Hauptspeicher mit Hilfe der malloc-Funktion angefordert werden, was normalerweise vermieden wird, da die Simulatoren des DISQUE-Testbetts eigene Freispeicherlisten für Ereignis- und Kundendatenstrukturen führen.

In Abb. 6.13 wurden die Ausführungszeiten sämtlicher malloc-Aufrufe für die Erzeugung neuer Ereigniseinträge bei den jeweils kleinsten `tandem`- und `maxagraphI`-Modellen (beide bestehen aus 70 Warteschlangen) gemessen und in ihrer zeitlichen Reihenfolge dargestellt. Wie man sieht, tauchen in regelmäßigen Abständen extrem teure malloc-Aufrufe (gemessen am Maßstab 100 μs) auf, deren Dauer obendrein noch linear mit der Anzahl der bereits erfolgten malloc-Aufrufe ansteigt. Die genaue Ursache dieses Phänomens konnte leider nicht ermittelt werden, da der Quellcode der verwendeten C-Bibliothek nicht zur Verfügung stand, man kann aber vermu-

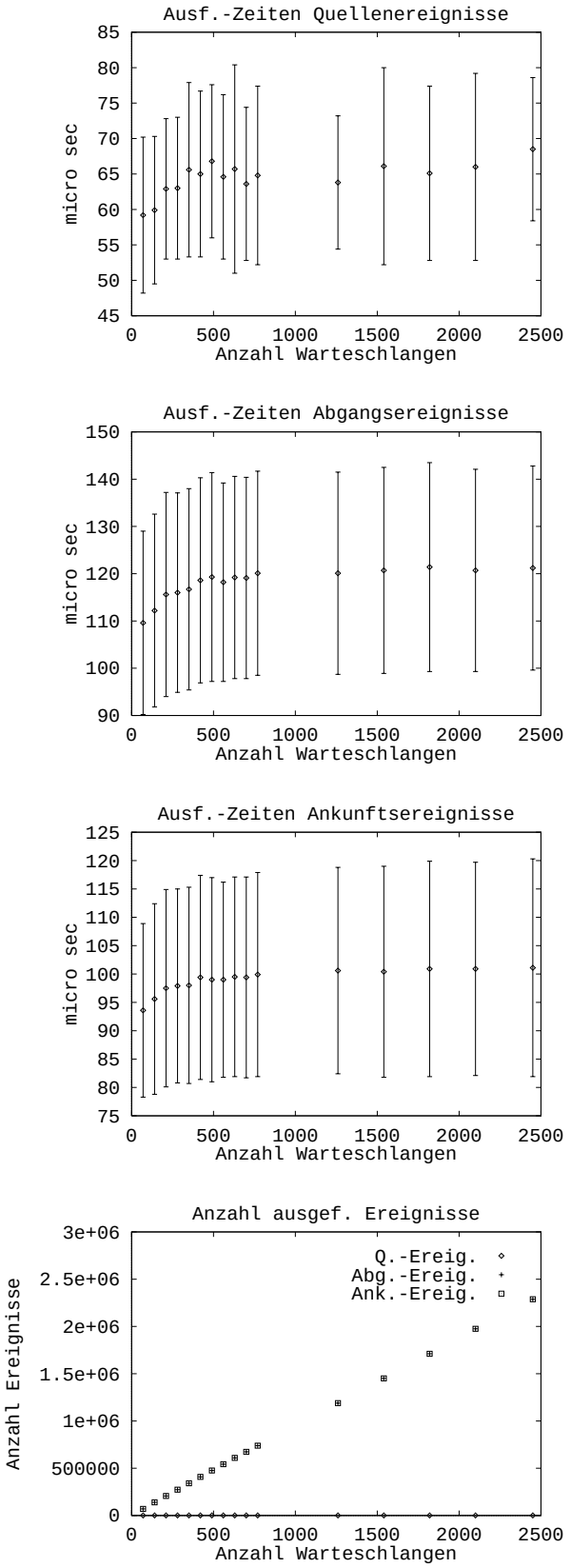


Abbildung 6.6: Ereignisstat. `tandem`

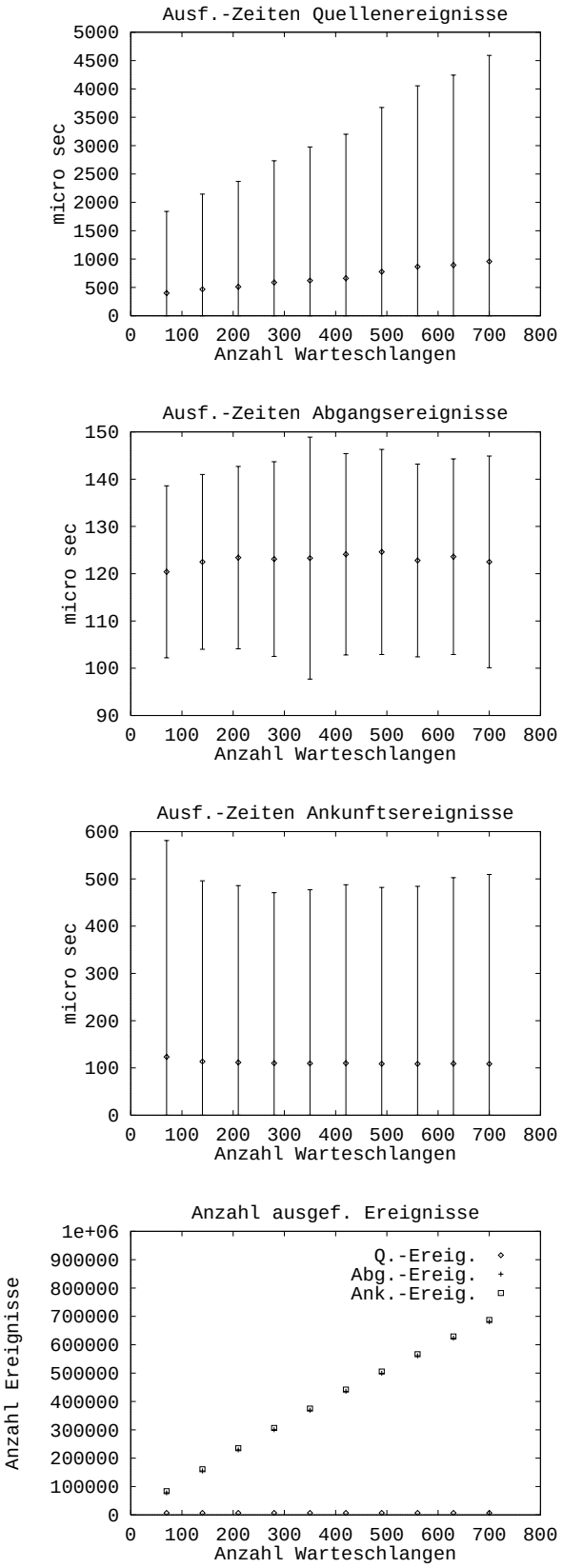


Abbildung 6.7: Ereignisstat. `maxagraphI`

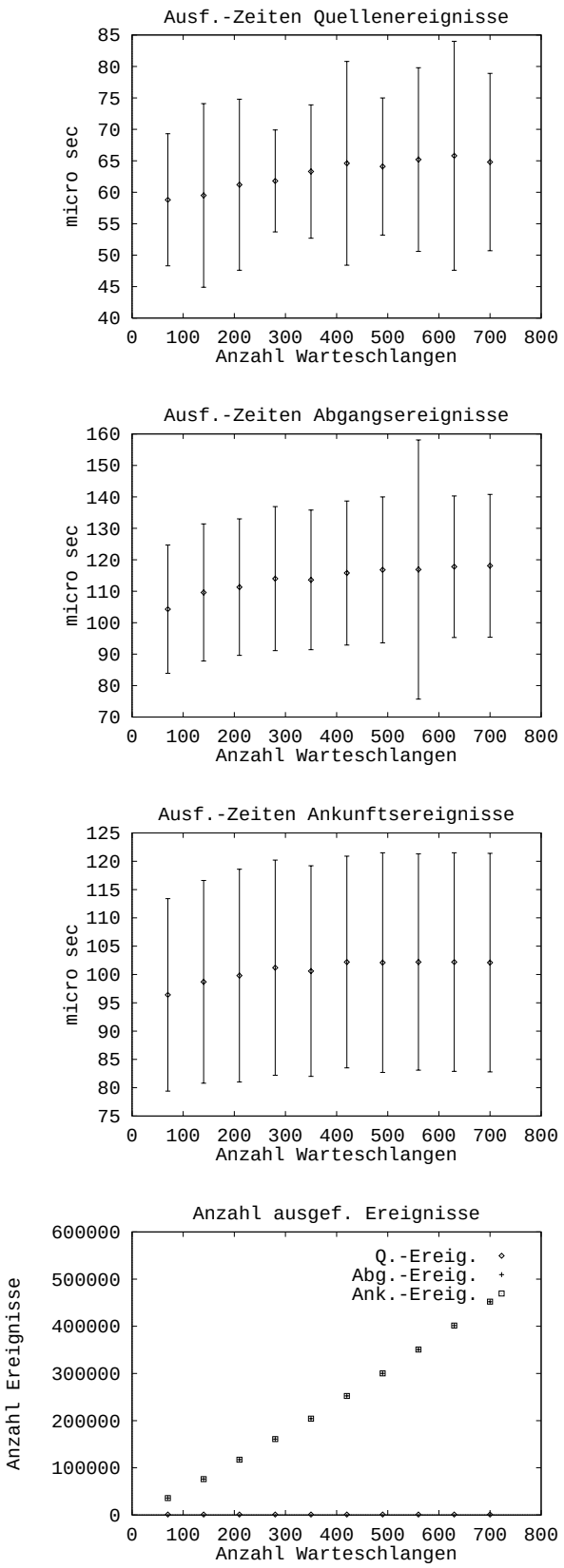


Abbildung 6.8: Ereignisstat. `maxagraphII`

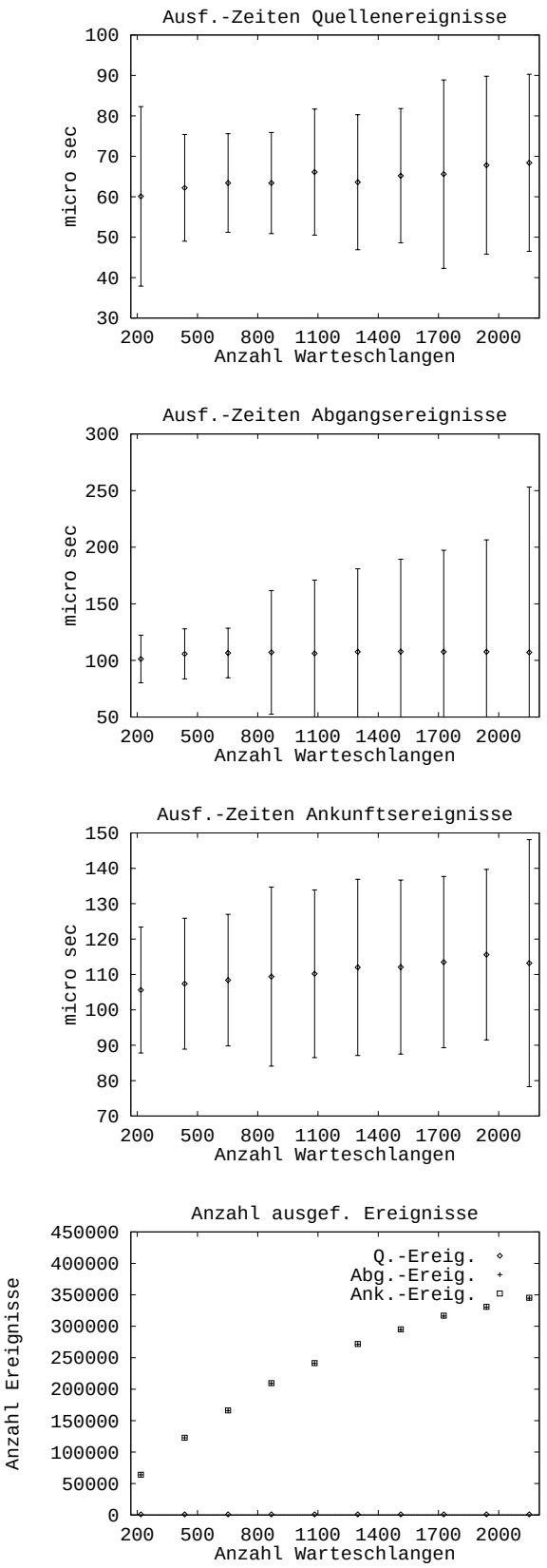


Abbildung 6.9: Ereignisstat. `maxagraphIII`

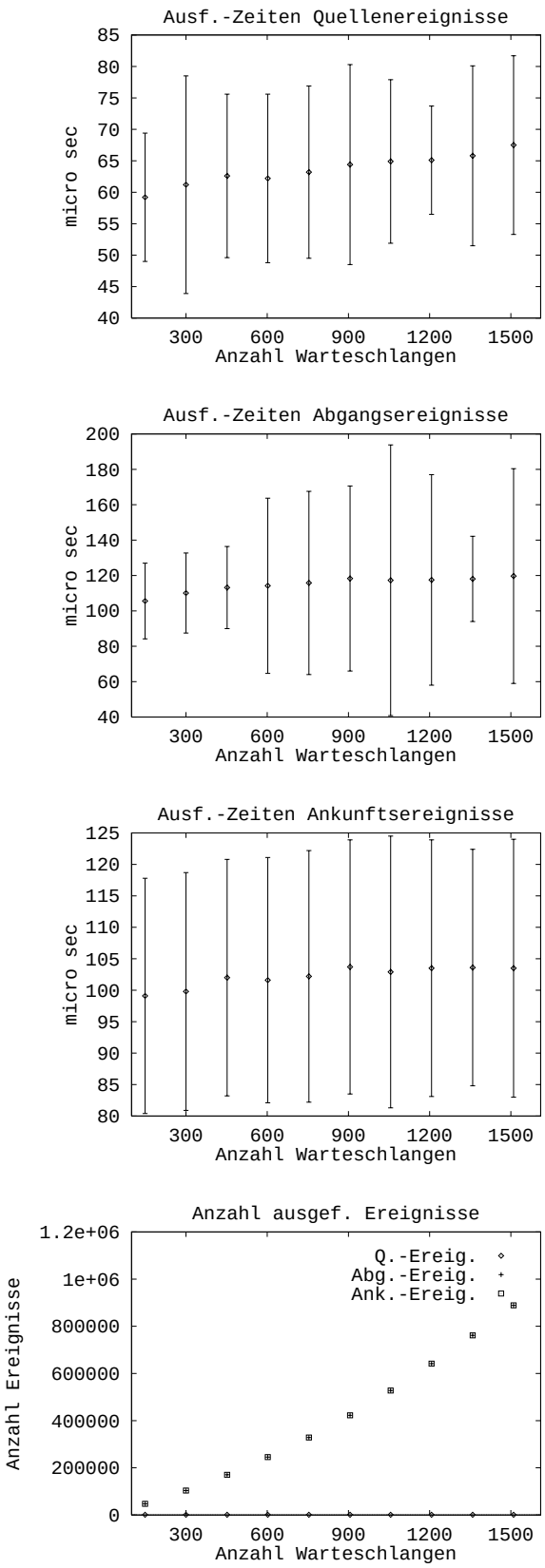


Abbildung 6.10: Ereignisstat. `branchI`

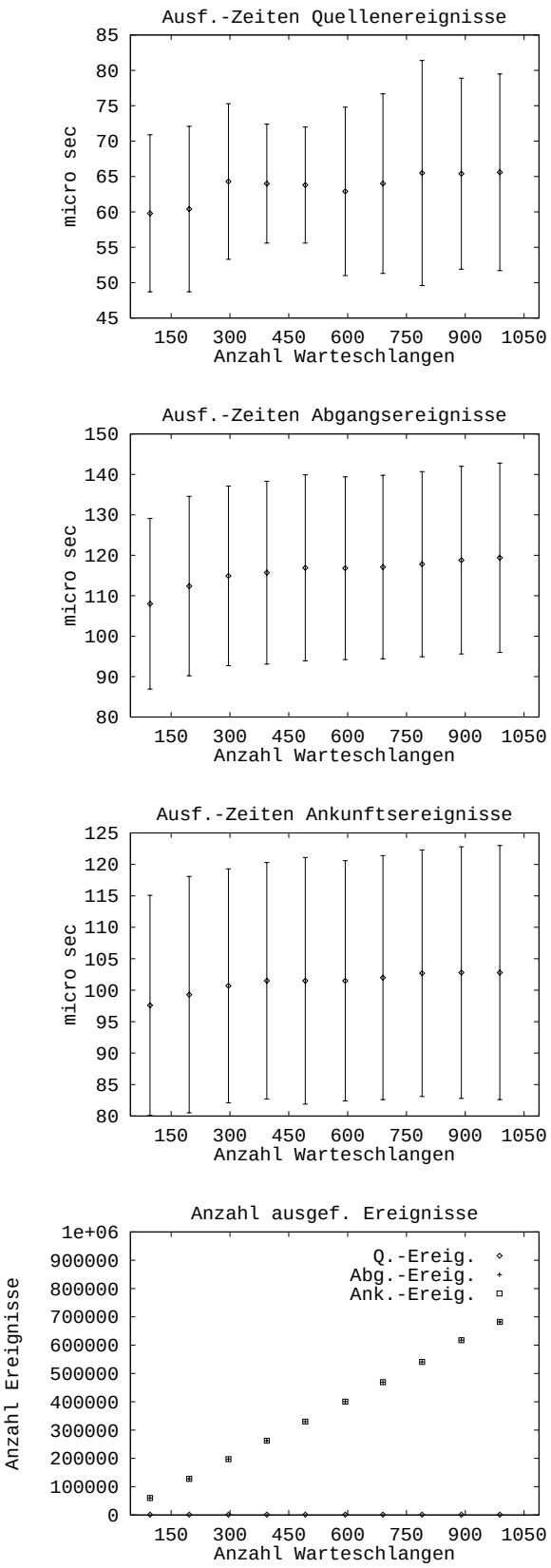


Abbildung 6.11: Ereignisstat. `branchII`

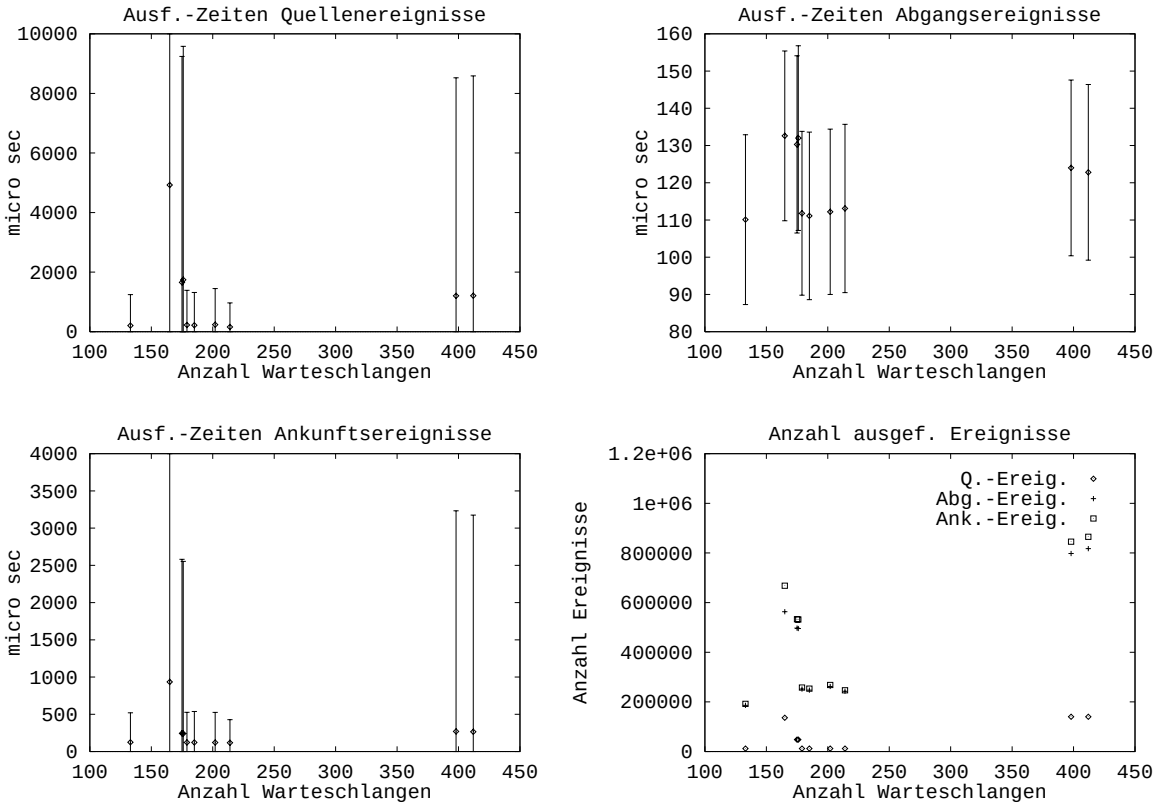


Abbildung 6.12: Ereignisstat. `iscas`

ten, daß es sich dabei um eine ineffektive Verwaltung größerer Hauptspeicherblöcke durch die malloc-Funktion (z. B. über lineare Listen) handelt. Bei der Simulation der `maxagraphI` und `iscas`-Modelle führt dies dazu, daß mehr teure malloc-Aufrufe als sonst bei der Ausführung von Ereignissen auftreten und diese obendrein noch gegen Ende der Simulation deutlich teurer als die teuren Aufrufe bei anderen Modellreihen werden.

6.2 Zusätzlicher Aufwand durch Parallelisierung des Simulationsmodells

Im folgenden werden die Laufzeiten der Simulatoren `optsim` und `simulate` miteinander verglichen. Wie in 5.2.1 bereits erwähnt, handelt es sich dabei in beiden Fällen um sequentielle Simulatoren, wobei jedoch bei `optsim` im Gegensatz zu `simulate` der Abgang eines Kunden und seine Ankunft an der nächsten Warteschlange zu

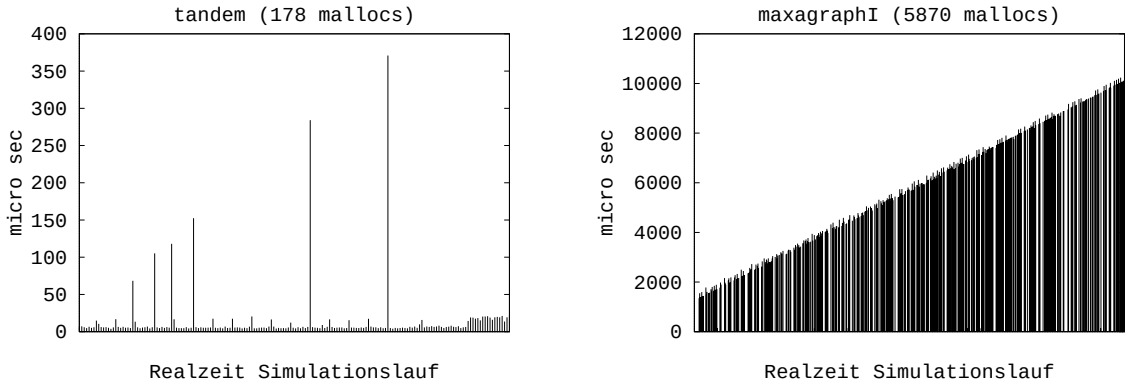


Abbildung 6.13: Dauer von malloc-Aufrufen beim Erzeugen neuer Ereignisse

einem gemeinsamen Ereignis zusammengefaßt sind, was bei einer parallelen Simulation nicht möglich wäre. Es geht also um den Mehraufwand durch Änderungen am Simulationsmodell, der für eine Zerlegung desselben in Teilmodelle mit disjunkten Zustandsräumen erforderlich ist.

Für jedes der in 6.1 beschriebenen Modelle wurden dazu folgende Größen gemessen:

t_{opt} : Laufzeit von `optsim`
t_{sim} : Laufzeit von `simulate`
t_{opt}^{elz} : Summe der Dauer von Ereignislistenzugriffen bei der Simulation mit `optsim`
t_{sim}^{elz} : Summe der Dauer von Ereignislistenzugriffen bei der Simulation mit `simulate`

Abb. 6.14 zeigt den Wert des Quotienten $\frac{t_{sim}-t_{opt}}{t_{opt}}$. Er gibt an, um welchen Anteil der Laufzeit von `optsim` sich die Laufzeit der sequentiellen Simulation des Warteschlangennetzes bei der Verwendung von `simulate` an Stelle von `optsim` verlängert. Wie man Abb. 6.14 entnehmen kann, liegt dieser Wert bei ca. 20%, was unter anderem bedeutet, daß bereits bei einer Beschleunigung durch parallele Simulation von 1.2 gegenüber der Laufzeit von `simulate` dieser zusätzliche Aufwand ausgeglichen wird. Die Anpassung der Simulationsmodelle an die besonderen Voraussetzungen für die parallele Simulation stellt daher bei den hier untersuchten Modellen kein ernsthaftes Hindernis für das Erzielen von Beschleunigungen durch parallele Simulation dar.

Interessant ist in diesem Zusammenhang noch, woher der zusätzliche Rechenaufwand bei `simulate` herrührt. Abb. 6.15 zeigt dazu den Quotienten $\frac{t_{sim}^{elz}-t_{opt}^{elz}}{t_{opt}}$. Dieser gibt an, um wieviel die zusätzlich nötigen Ereignislistenzugriffe bei `simulate` dessen Laufzeit gegenüber `optsim` verlängern. Wie man Abb. 6.15 entnehmen kann, liegt

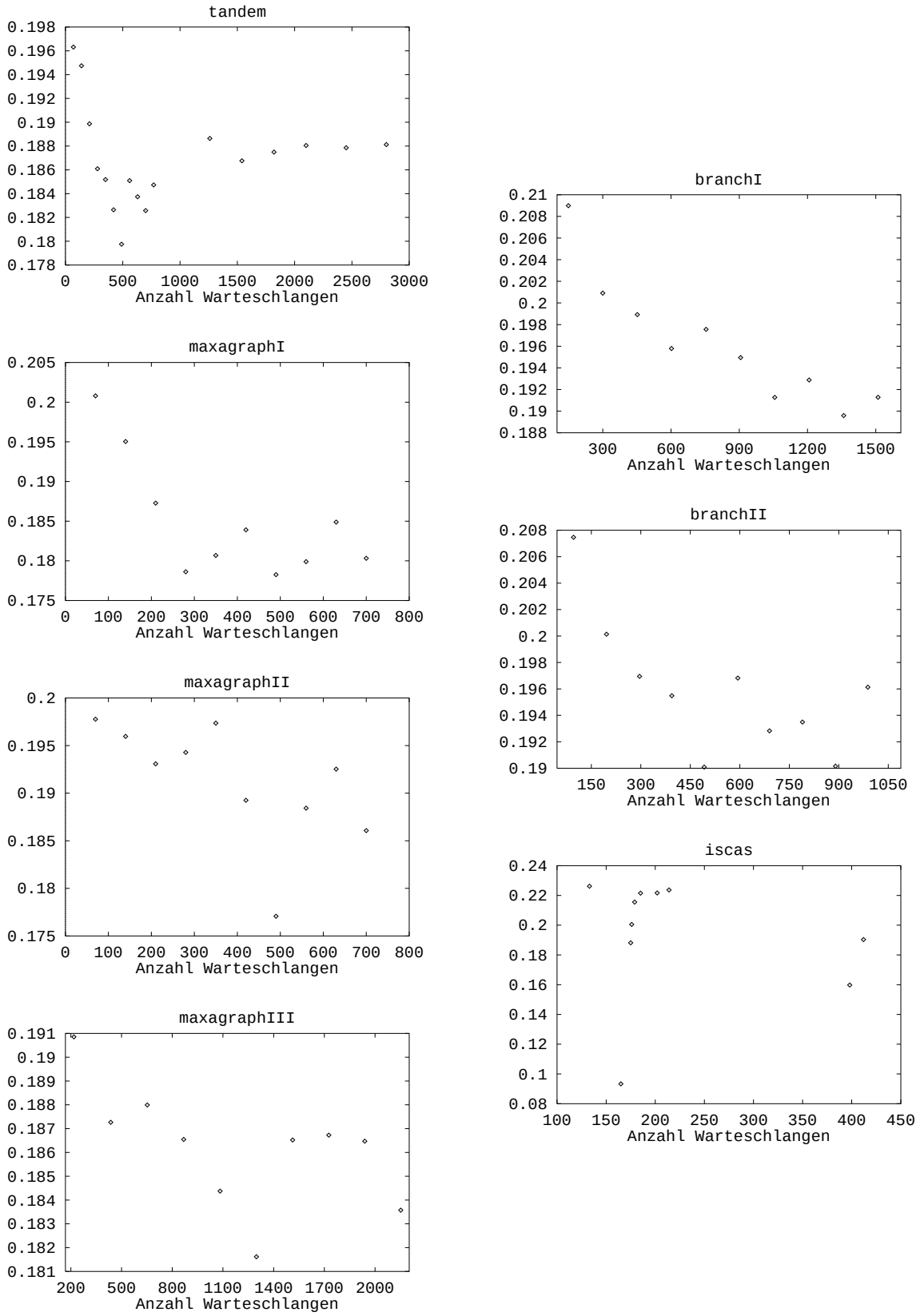


Abbildung 6.14: $\frac{t_{sim}-t_{opt}}{t_{opt}}$ für verschiedene Modelle

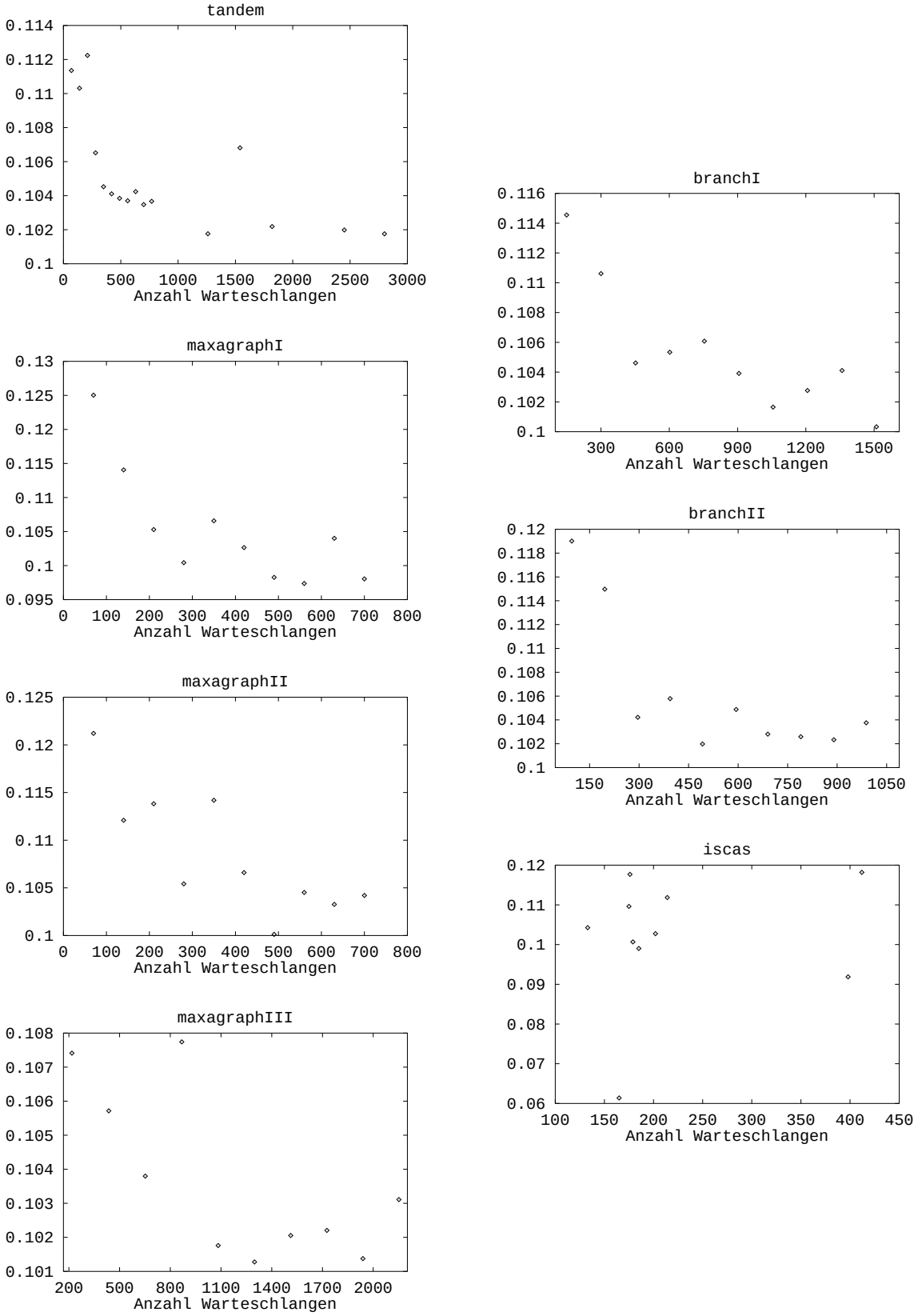


Abbildung 6.15: $\frac{t_{sim}^{elz} - t_{opt}^{elz}}{t_{opt}}$ für verschiedene Modelle

dieser Wert bei ca. 12%, was bedeutet, daß die zusätzlichen Zugriffe nur etwa die Hälfte des Mehraufwands ausmachen. Dies ist zunächst recht erstaunlich, da sowohl `simulate` als auch `optsim` denselben Code (d. h. dieselben C-Funktionen) bei Abgang und Ankunft eines Kunden aufrufen, der einzige Unterschied zwischen ihnen also die Verteilung des Codes auf zwei Ereignisse (bei `simulate`) bzw. ein gemeinsames Ereignis (bei `optsim`) ist. Nähere Untersuchungen ergaben, daß zusätzlicher Aufwand dadurch entsteht, daß die Parameter für die betreffenden Funktionen bei `simulate` im Gegensatz zu `optsim` nicht in lokalen Variablen bereits vorhanden sind, sondern erst aus einem Ereignis und damit aus dem Heap des Prozessorknotens gewonnen werden müssen. Messungen an den `iscas`-Modellen ergaben beispielsweise, daß allein der Aufwand für Anfordern und Freigeben des zusätzlich nötigen Ankunftsereignisses sowie der Transfer der Parameter bei `simulate` zwischen 2.5% und 3.5% der Laufzeit von `optsim` ausmachte! Dies legt die Vermutung nahe, daß bei der Parallelisierung von Simulationsmodellen, die erheblich stärker für die sequentielle Simulation ausoptimiert wurden (z. B. indem bei der Ausführung von Ereignissen auf große Teile des Gesamtzustands des Simulators zugegriffen wird), der zusätzlich entstehende Rechenaufwand erheblich höher ausfallen könnte[1].

Ein Nebenaspekt der hier durchgeführten Messungen war, daß diese auch gleich dazu benutzt werden konnten, die Komplexität der Zugriffe auf die in den Simulatoren des DISQUE-Testbetts als Ereignislisten benutzten Datenstrukturen zu untersuchen. Hierbei handelt es sich um die von Brown in [Bro88] beschriebenen "Calendar-Queues". Die Laufzeit der Zugriffszeiten werden dort als $O(1)$ angegeben, d. h. insbesondere beschränkt und unabhängig von der Länge der Ereignisliste. Brown führt den Nachweis allerdings nur experimentell mit künstlich erzeugten Ereignislistenzugriffen, indem er auf einer ebenfalls künstlich gefüllten Calendar-Queue eine größere Anzahl von sogenannten hold-Operationen (Entfernen des vordersten und Einfügen eines neuen Ereignisses) durchführt. In [CSR93] wurden diese Experimente in allgemeinerer Form mit einem auf Markov-Ketten beruhenden stochastischen Modell für die künstlichen Ereignislistenzugriffe sowie mit einer realen Anwendung (Simulation eines Rechnernetzes) durchgeführt. Untersucht wurden dabei verschiedene Implementierungen von Ereignislisten (lineare Listen, Splay Trees, etc.), unter denen sich auch die Calendar-Queues befanden. Sie schnitten dort zwar sehr gut ab, falls ihre Länge in etwa konstant blieb, reagierten aber mit zum Teil recht starken Schwankungen der Zugriffszeiten, falls dies nicht der Fall war. Insgesamt erwies sich keine

[1] Ein interessantes Beispiel für ein ähnlich gelagertes Problem findet man in [WRJ93]. Bei der dort beschriebenen Gefechtsfeldsimulation hängt das Laufzeitverhalten der Simulationen stark von der Anzahl der Sektoren ab, in die das Gefechtsfeld aufgeteilt ist, wobei die optimalen Anzahlen für sequentielle und parallele Simulation verschieden sind. Die sequentielle Simulation läuft mit der für die parallele Simulation optimalen Anzahl von Sektoren 2.3 mal langsamer als mit der optimalen Anzahl für sequentielle Simulation. Hierbei handelt es sich also um zwei parallele Simulationsmodelle, die sich unterschiedlich gut für sequentielle Simulation eignen.

der dort untersuchten Implementierungen als in allen Situationen überlegen.

Abb. 6.16 und 6.17 zeigen die mit sämtlichen Modellen bei der Simulation mit `optsim` und `simulate` gemessenen Werte, getrennt nach enqueue- (Einfügen eines Ereignisses) und dequeue- (Entfernen des Ereignisses mit dem kleinsten Zeitstempel der Ereignisliste) Zugriffen. Jedem simulierten Modell entspricht ein Punkt mit dem Mittelwert der Zugriffszeit und dem Mittelwert der Ereignislistenlänge beim Zugriff als Koordinaten, der von einem Intervall mit der Länge der doppelten Standardabweichung der Zugriffszeit umgeben ist. Wie man sieht, ist der Mittelwert der Zugriffszeit sowohl bei enqueue als auch bei dequeue annähernd konstant, jedoch streuen die einzelnen Zugriffszeiten in Abängigkeit vom simulierten Modell zum Teil ziemlich stark, wobei zwischen der Stärke der Streuung und der Länge der Ereignisliste kein Zusammenhang erkennbar ist. Im vorliegenden Fall ist also die Zugriffszeit auf die Calendar-Queues zumindest im Mittel konstant, so daß eine deutliche Veränderung der Zugriffszeiten bei paralleler Simulation eher unwahrscheinlich ist.

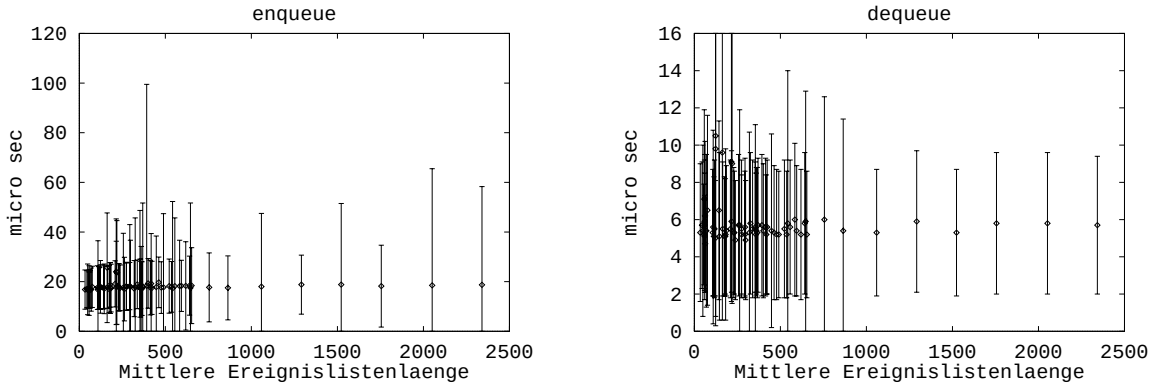


Abbildung 6.16: Dauer von enqueue/dequeue-Operationen bei `simulate`

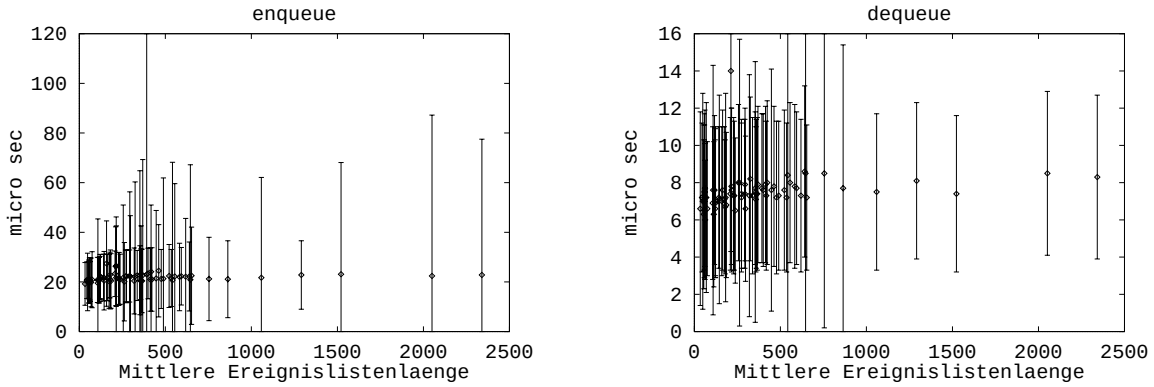


Abbildung 6.17: Dauer von enqueue/dequeue-Operationen bei `optsim`

6.3 Kritische-Pfad-Analyse

Mit den im folgenden vorgestellten Ergebnissen der Analyse der Lastbalancierung und der Kritischen-Pfad-Analyse (vgl. 3.4.1) wird der Frage nachgegangen, in wieweit schon die Struktur des verteilten Simulationsmodells der mit paralleler Simulation erzielbaren Beschleunigung Grenzen setzt, selbst unter Vernachlässigung von Kommunikations- und Synchronisationsaufwand. Bei den Messungen waren die Warteschlangen wie in 6.1 beschrieben zu Teilmodellen geclustert. Für jedes Teilmodell wurde ein Simulator auf einem eigenen Prozessor vorausgesetzt.

Folgende Größen wurden mit Hilfe von `cpasim` gemessen:

b_{last} : Ergebnis der Lastanalyse (d. h. die maximal mit paralleler Simulation erzielbare Beschleunigung, wenn alle Ereignisse bereits zu Beginn der Simulation bekannt wären)

b_{kpa} : Ergebnis der Kritischen-Pfad-Analyse (d. h. die mit paralleler Simulation erzielbare Beschleunigung, wobei im Unterschied zur Lastanalyse Ereignisse erst nach ihrer Erzeugung ausgeführt werden)

n : Anzahl der Teilmodelle, in die das Warteschlangennetz aufgeteilt wurde (6 für die `branchI`- und 7 für alle anderen Modelle)

Da grundsätzlich $b_{kpa} \leq b_{last} \leq n$ gilt, wurden die Ergebnisse beider Analysen durch n dividiert (analog zu dem in der Literatur für Beschleunigungsmessungen verwendeten Wert "Effizienz"). Die so entstandenen Zahlen zeigen, welcher relative Anteil des maximal möglichen Wertes erreicht wurde, was insbesondere den Vergleich von Modellen mit unterschiedlichem n ermöglicht.

Abb. 6.18 zeigt die Werte für alle untersuchten Modelle. Wie man deutlich sieht, sind sie bei einigen Modellen deutlich kleiner als 1. Sehr gute Werte liefern erwartungsgemäß die `tandem`-Modelle, bei denen, wie ebenfalls zu erwarten war, $b_{last} \approx b_{kpa}$ gilt, was zeigt, daß die kausalen Abhängigkeiten zwischen den Ereignissen so gut wie keine Rolle spielen. Dies ist des weiteren wohl auch typisch für alle Modelle mit schlechter Lastbalancierung, wie beispielsweise die Werte für die `maxagraphII`-Modelle zeigen und hat damit zu tun, daß überlastete Simulatoren mit der Arbeit nicht nachkommen (und damit der kritische Pfad bei der Bestimmung von t_{kpa} aus genau diesen Ereignissen besteht) und daher als letzte mit der Arbeit fertig werden (und so das letzte von ihnen ausgeführte Ereignis den Wert von t_{bal} bestimmt).

Eine gewisse, wenn auch eher geringe Rolle spielen die kausalen Abhängigkeiten bei den anderen Modellen, wie man an den abweichenden Werten für b_{last} und b_{kpa} erkennen kann. Allerdings wurden sie auch gezielt daraufhin entwickelt. Typisch für praktische Probleme sind wohl eher die `iscas`-Modelle, bei denen b_{last} und b_{kpa} fast identisch sind, d. h. die erzielbare Beschleunigung vom Modell her nur durch die ungleichmäßige Lastbalancierung begrenzt wird. Hier zeigt sich im übrigen auch ein zentrales Problem bei der zyklenfreien Partitionierung der Warteschlangennetze: Dadurch, daß Last immer nur in Form von ganzen Zusammenhangskomponenten auf einzelne Simulatoren verteilt werden kann, ist Lastbalancierung nur eingeschränkt möglich, was bei manchen `iscas`-Modellen, bei denen einige wenige Zusammenhangskomponenten einen Großteil der Last enthalten, gleichmäßige Lastverteilung unmöglich macht. Andererseits zeigen die Werte der `iscas`-Modelle auch, daß es durchaus Fälle gibt, bei denen sich die Last gleichmäßig verteilen läßt.

Zusammenfassend kann man also sagen, daß die durch parallele Simulation maximal erzielbare Beschleunigung durch ungleichmäßige Lastverteilung und kausale Abhängigkeiten bei einigen Modellen recht stark (teilweise auf weniger als 50% der vorhandenen Teilmodelle) eingeschränkt wird, wobei das dominierende Problem die gleichmäßige Verteilung der Last ist.

Abschließend sei noch an einem Beispiel demonstriert, warum Kritische-Pfad-Analyse ohne Berücksichtigung des Clusterns in Teilmodelle keinen Sinn macht. Abb. 6.19 zeigt die Ergebnisse der Analyse der Lastbalancierung und der Kritischen-Pfad-Analyse für die `maxagraphI`- und `maxagraphII`-Modelle unter der Annahme, daß jede Warteschlange ein separates Teilmodell darstellt und jeder Simulator eines Teilmodells seinen eigenen Prozessor besitzt. Wie man sieht, liegen die Werte von $\frac{t_{last}}{n}$ und $\frac{t_{kpa}}{n}$ unter 50%, wobei die Werte der `maxagraphII`-Modelle erheblich besser sind. Obwohl dies sogar für den Vergleich der absoluten Werte t_{kpa} und t_{last} gilt, wäre es, wie bereits gezeigt wurde, trotzdem falsch, daraus auf einen höheren "potentiellen Parallelismus" der `maxagraphII`-Modelle gegenüber den `maxagraphI`-Modellen zu schließen. Dies liegt daran, daß im Regelfall bei einem großen Warteschlangennetz die gesamte Simulationslast unterschiedlich auf die einzelnen Warteschlangen verteilt ist. Die in Abb. 6.19 dargestellten Werte sagen somit lediglich etwas über die ungleiche Verteilung der Last auf die einzelnen Warteschlangen aus (die bei den `maxagraphI`-Modellen größer ist, da dort für die Warteschlangen, die eine Kundenquelle besitzen, mehr Ereignisse ausgeführt werden müssen als bei anderen Warteschlangen). Ziel der Lastbalancierung ist es aber gerade, solche Unterschiede durch Zusammenfassen einer "Mischung" von Warteschlangen mit hoher und niedriger Last auszugleichen. Folglich macht es keinen Sinn, von dem "potentiellen Parallelismus" eines Warteschlangennetzes zu sprechen, solange mit diesem nicht eine Aufteilung desselben in Teilmodelle verbunden ist.

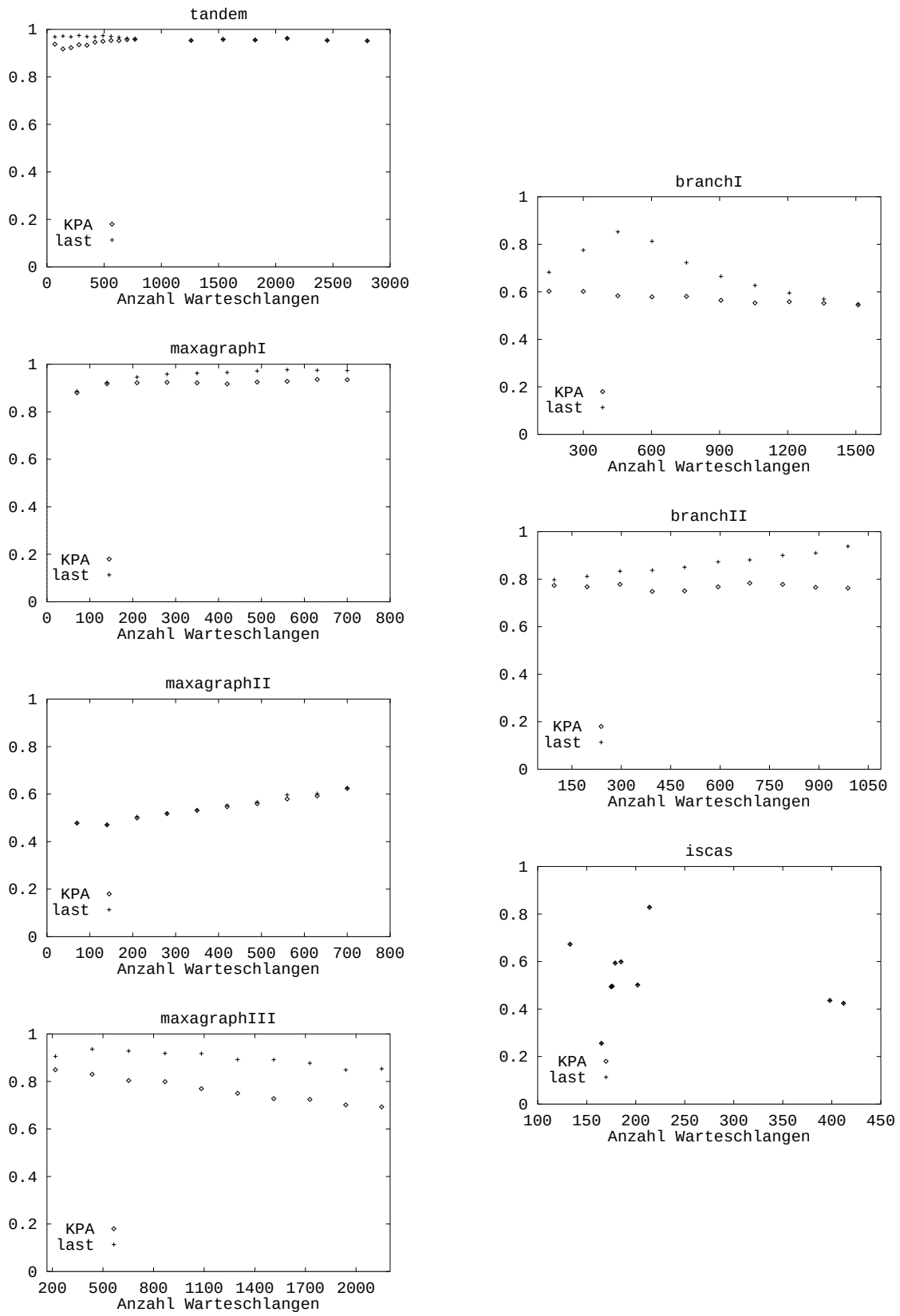


Abbildung 6.18: $\frac{b_{kpa}}{n}$ und $\frac{b_{last}}{n}$ für verschiedene Modelle

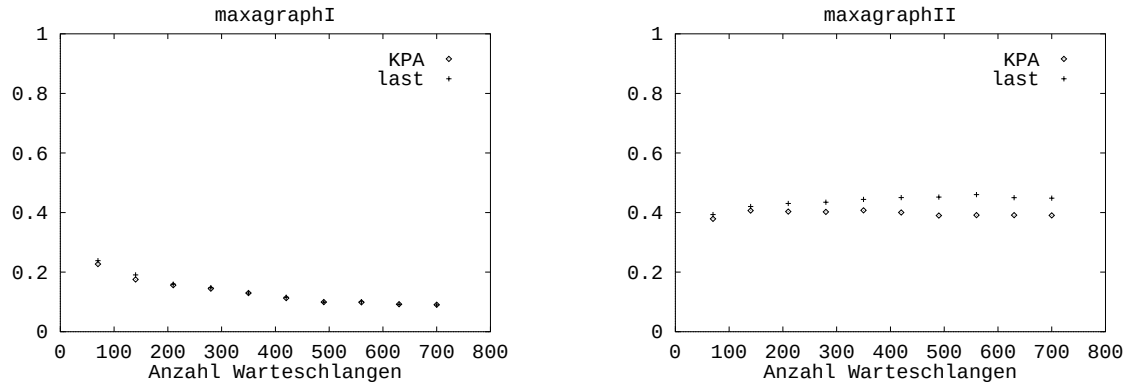


Abbildung 6.19: $\frac{b_{kpa}}{n}$ und $\frac{b_{last}}{n}$ ohne Berücksichtigung des Clusterns

6.4 Zusätzlicher Rechenaufwand durch Parallelisierung des Simulatorcodes

In diesem Abschnitt wird der zusätzliche Aufwand untersucht, der dadurch entsteht, daß der Code eines sequentiellen Simulators an die besonderen Gegebenheiten eines parallelen Systems, insbesondere im Hinblick auf Synchronisation und Kommunikation, angepaßt werden muß. Dazu wird die Laufzeit von `simulate` mit der eines parallelen Simulationslaufs mit nur einem Simulatorknoten verglichen. Da bei letzterem sämtliche Warteschlangen zu einem einzigen sequentiellen Simulationsmodell verschmolzen werden, arbeitet der Simulatorprozeß dann im Prinzip wie ein sequentieller Simulator, da er nie auf das Eintreffen eines weiteren Ereignisses warten muß. Letztendlich handelt es sich also um einen sequentiellen Simulationslauf, bei dem jedoch alle für die Ausführung eines Ereignisses durch den parallelen Simulator notwendigen Aktionen durchgeführt werden. Um den Aufwand für das (im Prinzip überflüssige) Synchronisationsverfahren so gering wie möglich zu halten, wurde dafür das Linktime-Verfahren (vgl. 3.2) verwendet.

Gemessen wurden die folgenden Größen:

t_{clu} : Laufzeit des parallelen Simulators mit einem einzigen Simulatorknoten

t_{sim} : Laufzeit des sequentiellen Simulators `simulate`

Abb. 6.20 zeigt den Wert von $\frac{t_{clu}}{t_{sim}}$ für die verschiedenen Modellreihen, bei denen leider auf Grund von Speicherplatzproblemen der Wert von t_{clu} nicht für alle Modelle gemessen werden konnte. Von einigen Ausnahmen abgesehen, liegen die Werte

durchweg zwischen 4.0 und 5.0, was unter Berücksichtigung der Ergebnisse des vorherigen Abschnittes bedeutet, daß auf Grund des Mehraufwandes, der durch die Parallelisierung bedingt ist, bei der parallelen Simulation der Modelle mit nur 7 Knoten keine nennenswerten Beschleunigungen möglich sind. Bei der Suche nach der Ursache dieses hohen, zusätzlichen Aufwandes gilt der erste Gedanke naturgemäß den bei der Programmierung eines parallelen Systems zusätzlich notwendigen Operationen für Synchronisation und Kommunikation. Die folgende Tabelle zeigt Mittelwert und Standardabweichung der Laufzeiten der MMK-Operationen reqsema (Warten an einem Semaphor), relsema (Freigeben eines Semaphors), disable (Ausschalten des Interrupts für den Scheduler der Threads), enable (Einschalten des Schedulers), sndmsg (Versenden einer Nachricht) und recmsg (Empfang einer Nachricht). Sämtliche Operationen wurden dazu mit einem extra dafür geschriebenen Programm isoliert 1000000-mal ausgeführt, wobei sich bei den Semaphor- und Nachrichten-Operationen der adressierte Semaphor bzw. die adressierte Mailbox auf demselben Rechnerknoten wie das aufrufende Programm befand und daher (durch eine im MMK-Kern vorhandene Optimierung) kein Aufwand für Nachrichten-Kommunikation entstand.

Operation	Laufzeit (μs)	Std. Abw.
reqsema	115.4	8.7
relsema	117.3	8.8
disable	42.7	4.4
enable	44.0	5.0
sndmsg	134.7	9.6
recmsg	120.7	9.2

Wie man sieht, sind die Operationen, gemessen an dem in 6.1 entwickelten Maßstab 100 μs für die Ausführung eines Ereignisses sehr teuer. Es zeigt sich hier, daß die verwendete Programmstruktur (Multithreading mit Synchronisation der Threads über Semaphore) für die feine Granularität der Anwendung schlecht geeignet war: Vor und nach der Ausführung eines Ereignisses muß der Simulatorprozeß auf den Semaphor `data_access` zugreifen, um sich mit den Leseprozessen zu synchronisieren (vgl. 5.2.2). Die Zeit, die er dafür benötigt, ist bereits doppelt so hoch wie die eigentliche Ausführungszeit des Ereignisses. Weiterhin müssen gewisse Zugriffe auf globale, aber Betriebssystem-interne Strukturen wie etwa der von der malloc-Funktion verwaltete Freispeicher durch Ausschalten des Schedulers mit disable vor unsynchronisierten Zugriffen geschützt werden. Derartige Synchronisationsoperationen sind beim Multithreading unabdingbar.

Trotzdem erklärt dies alles nur Werte zwischen 3.0 und 4.0 für $\frac{t_{clu}}{t_{sim}}$. Der Rest geht wohl zu Lasten von diversen kleineren Unterschieden bei der Ausführung der Ereignisse durch den parallelen Simulator gegenüber `simulate`, wie beispielsweise

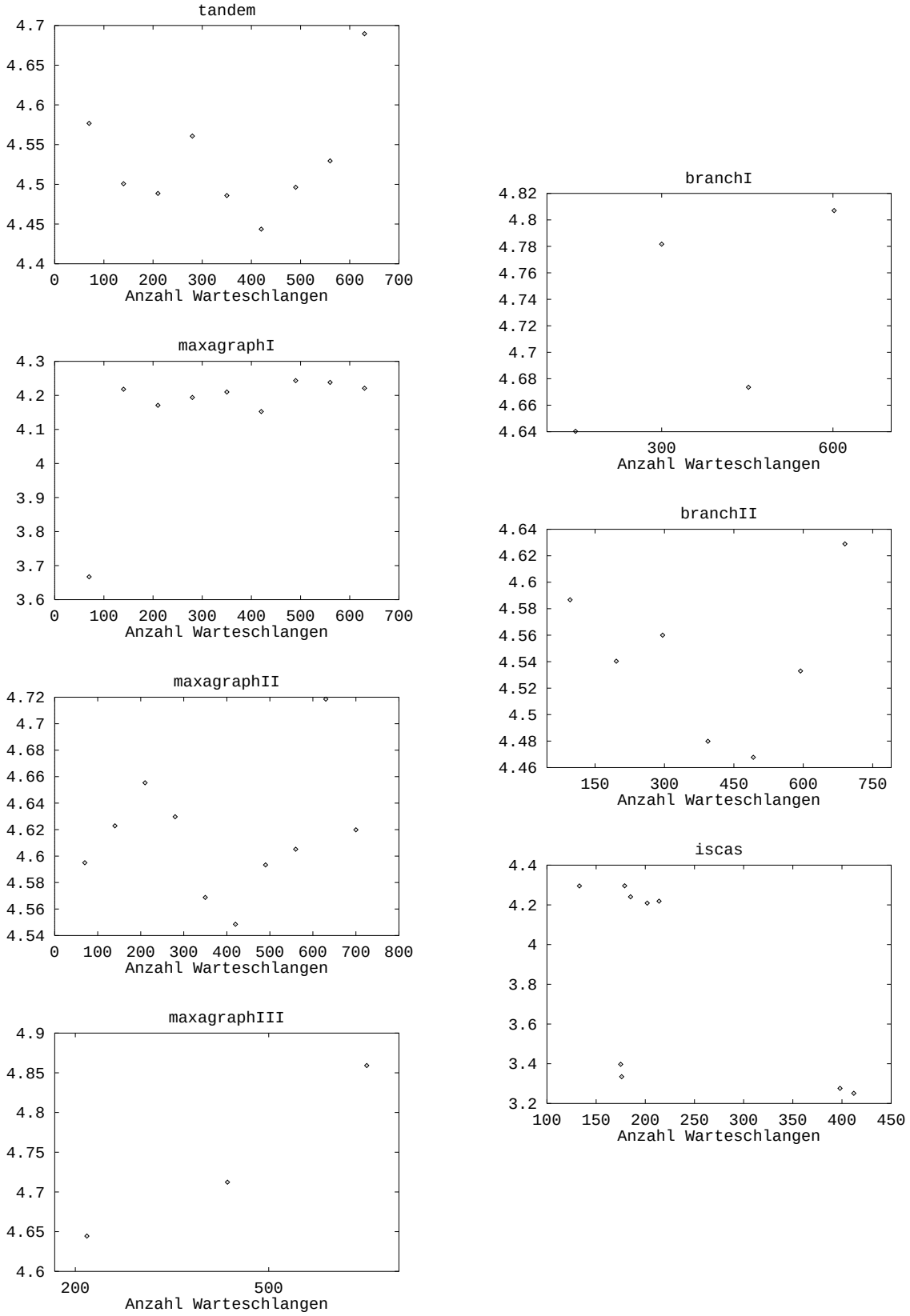


Abbildung 6.20: $\frac{t_{clu}}{t_{sim}}$ für verschiedene Modelle

den sim_filter und out_filter-Aufrufen (vgl. 5.2.2) oder etwa der Verwaltung von global eindeutigen Nummern für Ereignisse, die für die Erzeugung von Tracefiles notwendig sind.

Insgesamt läßt sich also sagen, daß der zusätzliche Aufwand bei der Ausführung von Ereignissen, der durch die Parallelisierung des Simulatorcodes entsteht, bedingt durch die feine Granularität der Anwendung ein sehr ernstes Problem darstellt. Bei echt paralleler Ausführung kommen noch zumindest die Kosten der Start-Zeiten für das Senden und Empfangen von Nachrichten dazu, die ebenfalls nicht unerheblich sind (vgl. sndmsg und recmsg in obiger Tabelle) und den Gesamtaufwand weiter deutlich erhöhen dürften. Berücksichtig man noch die Tatsache, daß diese Werte noch ungünstiger ausfallen, wenn man als sequentiellen Simulator `optsim` an Stelle von `simulate` zu Grunde legt, so kann man davon ausgehen, daß der zusätzliche Aufwand, der durch die Parallelisierung des Simulatorcodes entsteht, bei dem hier untersuchten System erst bei Einsatz von 6 Prozessoren "amortisiert" wird.

6.5 Analyse der verzögerten Ausführung von Ereignissen bei konservativen Synchronisationsverfahren

Im folgenden wird untersucht, wie stark sich die verzögerte Ausführung von Ereignissen, die durch den Einsatz konservativer Synchronisationsverfahren entsteht, auf die Gesamtlaufzeit paralleler Simulationsläufe auswirkt. Als Vergleichsmaßstab dient dabei ein mit Orakeldateien synchronisierter, paralleler Simulationslauf, bei dem, wie in 3.4.2 bereits näher erläutert wurde, jeder Simulator anhand einer aus einem früheren Simulationslauf gewonnenen Datei sofort ermitteln kann, ob er das vorderste Ereignis seiner Ereignisliste ausführen darf. Dies verursacht jedoch zusätzlichen Aufwand für Dateizugriffe, der in 6.5.1 untersucht wird. 6.5.2 beschäftigt sich mit der verzögerten Ausführung von Ereignissen beim Linktime-Verfahren, während in 6.5.3 der Frage nachgegangen wird, ob sich diese Verzögerung durch zusätzliches Versenden von Nullnachrichten reduzieren läßt. Schließlich wird in 6.5.4 die sich aus der Laufzeit des parallelen Simulators ergebende Beschleunigung gegenüber einem sequentiellen Simulationslauf mit den Ergebnissen der Kritischen-Pfad-Analyse aus 6.3 verglichen, um so der Frage nachzugehen, inwieweit deren Ergebnisse tatsächlich eine obere Schranke für die mit paralleler Simulation erzielbare Beschleunigung darstellen.

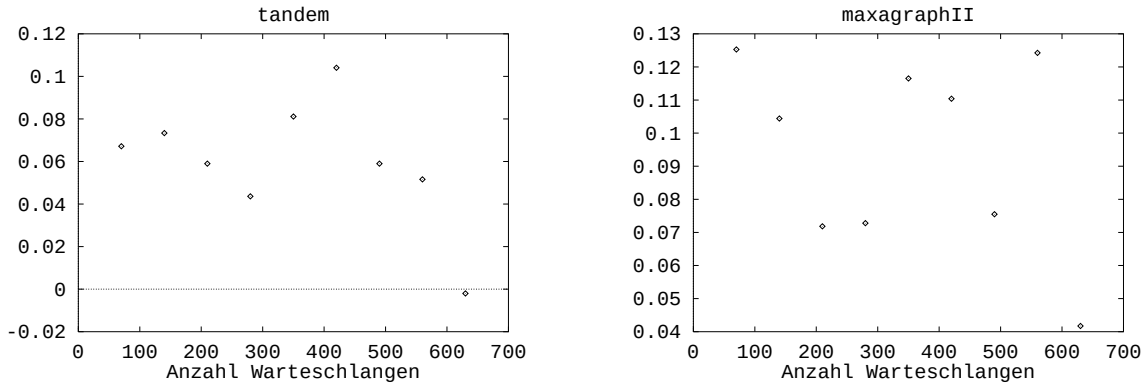


Abbildung 6.21: $\frac{t_{clu}^{ora} - t_{clu}}{t_{clu}}$ für `tandem` und `maxagraphII`

6.5.1 Zusätzlicher Aufwand für Dateizugriffe bei Orakeldateien

Wie bereits weiter oben erläutert wurde, geht es hier um den zusätzlichen Aufwand bei der Synchronisation paralleler Simulatoren mit Orakeldateien, der durch das Lesen der Dateien vom parallelen Dateisystem des iPSC860 (vgl. 5.1.3) entsteht. Dazu wurden analog zur Vorgehensweise in 6.4 die Laufzeiten zweier paralleler Simulationsläufe mit nur einem Simulatorknoten verglichen, wobei zum einen Linktime und zum anderen Orakeldateien als Synchronisationsverfahren verwendet wurden. Da das Linktime-Verfahren in diesem Fall praktisch keinen Zusatzaufwand verursacht, zeigt der Unterschied in der Laufzeit dann, wie sehr die Dateizugriffe die Ausführung von Ereignissen verlangsamen, wieviel also das Orakel kostet. Dies ist für die anschließenden Untersuchungen wichtig, da es einen Anhaltspunkt dafür gibt, wie stark sich die Synchronisation mit Orakeldateien von einem "idealen" Synchronisationsverfahren unterscheidet, bei dem jedes Ereignis auf jedem Simulatorknoten frühestmöglich ausgeführt wird.

Es wurden dazu folgenden Größen gemessen:

t_{clu} : Laufzeit des parallelen Simulators mit einem Simulatorknoten und Linktime als Synchronisationsverfahren

t_{clu}^{ora} : Wie t_{clu} nur mit Synchronisation über Orakeldateien

Abb. 6.21 zeigt den Wert von $\frac{t_{clu}^{ora} - t_{clu}}{t_{clu}}$ für die `tandem`- und `maxagraphII`-Modelle. (Alle anderen Modelle lieferten ähnliche Werte, wobei das Maximum und das Mi-

nimum bei den hier dargestellten Modellen auftrat.) Wie man sieht, verlängert der vor jedem Ereignis auszuführende Dateizugriff die Ereignisausführung um weniger als 13%, bei einem `tandem`-Modell verursachte er sogar weniger Aufwand als das Linktime-Verfahren, dessen einzige Aufgabe noch darin bestand, zu überprüfen, ob der Zeitstempel des auszuführenden Ereignisses das Simulationsende überschreitet. Dieser unerwartet gute Wert erklärt sich wohl einerseits aus den Besonderheiten des parallelen Dateisystems (paralleler Plattenzugriff mit Hilfe von spezialisierten Rechnerknoten, vgl. 5.1.3), wahrscheinlich aber andererseits auch daraus, daß die Orakeldateien nur für lesenden Zugriff geöffnet wurden und somit spezielle Optimierungen beim gepufferten I/O wie etwa das Füllen eines zweiten Puffers parallel zum Lesen des ersten durch das Betriebssystem NX2 möglich waren. Da der Quellcode des Dateisystems nicht zur Verfügung stand, konnte leider nicht überprüft werden, wie NX2 in solchen Fällen vorgeht.

Um diese Zahlen zu werten, ist es hilfreich, sich vor Augen zu halten, daß, wie in 6.2 dargelegt wurde, bereits einige wenige Hauptspeichermanipulationen eines Simulators die Ausführungszeit eines Ereignisses um mehr als 3% verlängern können. Somit entspricht der Mehraufwand für den Zugriff auf Orakeldateien z. B. in etwa dem Rechenaufwand für ein Synchronisationsverfahren, bei dem mehrfach auf Tabellen zugegriffen wird, ist aber natürlich deutlich höher als der Aufwand für extrem einfache Verfahren wie etwa Linktime. Die Synchronisation mit Orakeldateien stellt damit zwar kein "ideales" Synchronisationsverfahren dar, kommt einem solchen aber schon recht nahe (insbesondere, wenn man ca. 10% Zusatzaufwand für die Realisierung des Orakels herausrechnet). Es bietet damit eine gute Grundlage für die Bewertung anderer Verfahren bei Modellen, bei denen das Linktime-Verfahren nicht als Maßstab einsetzbar ist, da es zu nicht vorausberechenbaren Verzögerungen bei der Ausführung von Ereignissen führen kann.

6.5.2 Analyse des Linktime-Verfahrens

Bei dem hier untersuchten Linktime-Verfahren darf, wie in 3.2 bereits erläutert wurde, ein Ereignis immer dann ausgeführt werden, wenn sein Zeitstempel echt größer als das Minimum der Kanalzeiten ist. Da dieses Verfahren bis auf das Verwalten der Kanalzeiten keinen zusätzlichen Rechenaufwand und kein zusätzliches Nachrichtenaufkommen verursacht, gehört es zu den einfachsten Synchronisationsverfahren. (Es hat allerdings den Nachteil, daß es nur bei parallelen Simulatoren mit zyklenfreier Topologie der Simulatorknoten einsetzbar ist)

Folgende Größen wurden gemessen:

t_{link} : Laufzeit des parallelen Simulators mit Linktime als Synchronisationsverfahren

t_{ora} : Wie t_{link} nur mit Synchronisation über Orakeldateien

Im Gegensatz zum vorangegangenen Abschnitt wurden diese Laufzeiten bei echt parallelen Simulationsläufen mit 7 Simulatorknoten gemessen. Abb. 6.22 zeigt den Wert des Quotienten $\frac{t_{link}}{t_{ora}}$ für die untersuchten Modelle. Wie man deutlich sieht, wirkt sich die durch das Linktime-Verfahren bewirkte verzögerte Ausführung von Ereignissen nur bei den `maxagraphIII`-Modellen aus. Bei allen anderen Modellen sind die Werte von t_{link} entweder nur unwesentlich größer als die von t_{ora} oder aber in einigen Fällen sogar etwas kleiner, was wohl auf die bereits im vorangegangenen Abschnitt untersuchten Dateizugriffe bei der Synchronisation mit Orakeldateien zurückzuführen ist. Letzteres gilt insbesondere auch für die `branchI`- und `branchII`-Modelle, obwohl gerade diese Modelle so konstruiert wurden, daß die Kanalzeiten der einzelnen Eingangskanäle eines Simulatorknotens sich stark voneinander unterscheiden sollten.

Zusammenfassend kann man also sagen, daß die durch das Linktime-Verfahren bewirkte Verzögerung der Ereignisausführung bei zyklenfreier Topologie der Simulatorknoten sich zwar bei den hier untersuchten Modellen in Ausnahmefällen deutlich auswirkt, im allgemeinen aber kein großes Hindernis für die parallele Simulation darstellt.

6.5.3 Analyse konservativer Synchronisationsverfahren mit Nullnachrichten

Da es sich bei den hier untersuchten parallelen Simulationen ausschließlich um solche mit azyklischer Topologie der Simulatorknoten handelt, dient der Einsatz von Nullnachrichten lediglich dem schnelleren Hochsetzen der Kanalzeiten und nicht der Vermeidung von Deadlocks. Daher konnte zusätzlich zu dem in 3.2 geschilderten Verfahren (Versenden von Nullnachrichten nach jedem ausgeführten Ereignis über jeden Kanal, über den keine Ereignisnachricht versandt wurde) noch eine Variante untersucht werden, bei der die Ausführung eines Ereignisses nur dann das Verschicken von Nullnachrichten bewirkt, wenn dabei mindestens eine Ereignisnachricht für einen anderen Simulatorknoten erzeugt wurde. Dies hat insbesondere den Vorteil, daß eine Kette von Ereignissen, bei denen jedes einzelne Ereignis nur ein Folgeereignis für denselben Simulatorknoten erzeugt (z. B. wenn ein Kunde hintereinander mehrere leere Warteschlangen passiert, die zum selben Teilmodell gehören), nicht eine ganze Serie von Nullnachrichten erzeugt.

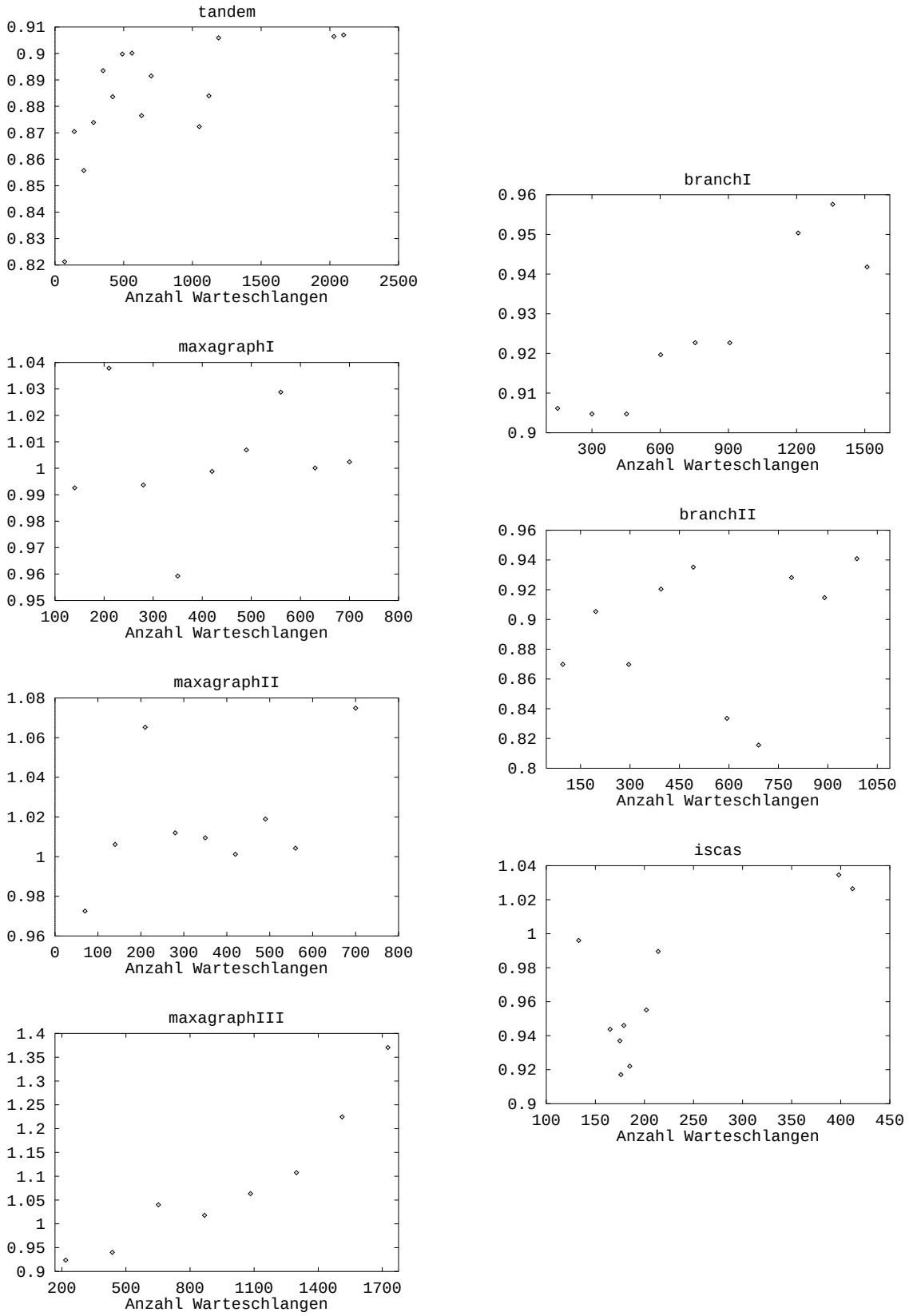


Abbildung 6.22: $\frac{t_{link}}{t_{ora}}$ für verschiedene Modelle

Gemessen wurden die folgenden Größen:

t_{link} : Laufzeit des parallelen Simulators mit Linktime als Synchronisationsverfahren

t_{avoid} : Wie t_{link} nur mit Deadlock-Avoidance als Synchronisationsverfahren

t_{avoid-} : Wie t_{link} nur mit reduziertem Versenden von Nullnachrichten

Abb. 6.23 zeigt den Wert des Quotienten $\frac{t_{avoid-}}{t_{link}}$ für die verschiedenen Modelle. Wie man sieht, führt das Verfahren zu keinen drastischen Laufzeitveränderungen gegenüber Linktime. Eine Ausnahme stellen die `maxagraphIII`-Modelle dar, bei denen sich die Laufzeit zum Teil deutlich verbessert, was im Einklang mit den im vorangegangenen Abschnitt dargestellten Werten von $\frac{t_{link}}{t_{ora}}$ steht. Eine andere Ausnahme stellen einige kleine `iscas`- und die kleinsten `maxagraphI`- und `maxagraphII`-Modelle dar, bei denen sich die Laufzeiten des parallelen Simulationslaufs deutlich verschlechtern, was wohl am hohen Verzweigungsgrad einiger Simulatorknoten verbunden mit häufigem Versenden von Ereignisnachrichten liegt.

Abb. 6.24 zeigt den Wert des Quotienten $\frac{t_{avoid}}{t_{link}}$. Man erkennt deutlich eine drastische Verschlechterung der Laufzeit durch das Versenden der Nullnachrichten gegenüber Linktime. Die fehlenden Werte einiger Modelle erklären sich dabei daher, daß die zugehörigen Simulationsläufe in den Bereich von Stunden (!) gerieten und daher vorzeitig abgebrochen wurden. Diese Zahlen zeigen noch einmal deutlich die Auswirkungen von Nullnachrichtenlawinen, wie sie in der Literatur bereits häufiger beschrieben wurden.

Insgesamt läßt sich also feststellen, daß bei den hier untersuchten Modellen Nullnachrichten, sparsam verwendet, die Laufzeit eines parallelen Simulationslaufs geringfügig, jedoch – von Ausnahmen abgesehen – nicht entscheidend verbessern konnten.

6.5.4 Vergleich der Laufzeiten des Linktime-Verfahrens und der Ergebnisse der Kritischen-Pfad-Analyse

Als Abschluß der Analyse konservativer Synchronisationsverfahren wird hier noch der Frage nachgegangen, in wieweit das Ergebnis der Kritischen-Pfad-Analyse tatsächlich eine obere Schranke für die mit paralleler Simulation erzielbare Beschleunigung darstellt. Hierzu wurden die Ergebnisse aus 6.3 mit den Beschleunigungen, die sich aus dem parallelen, mit dem Linktime-Verfahren synchronisierten Simula-

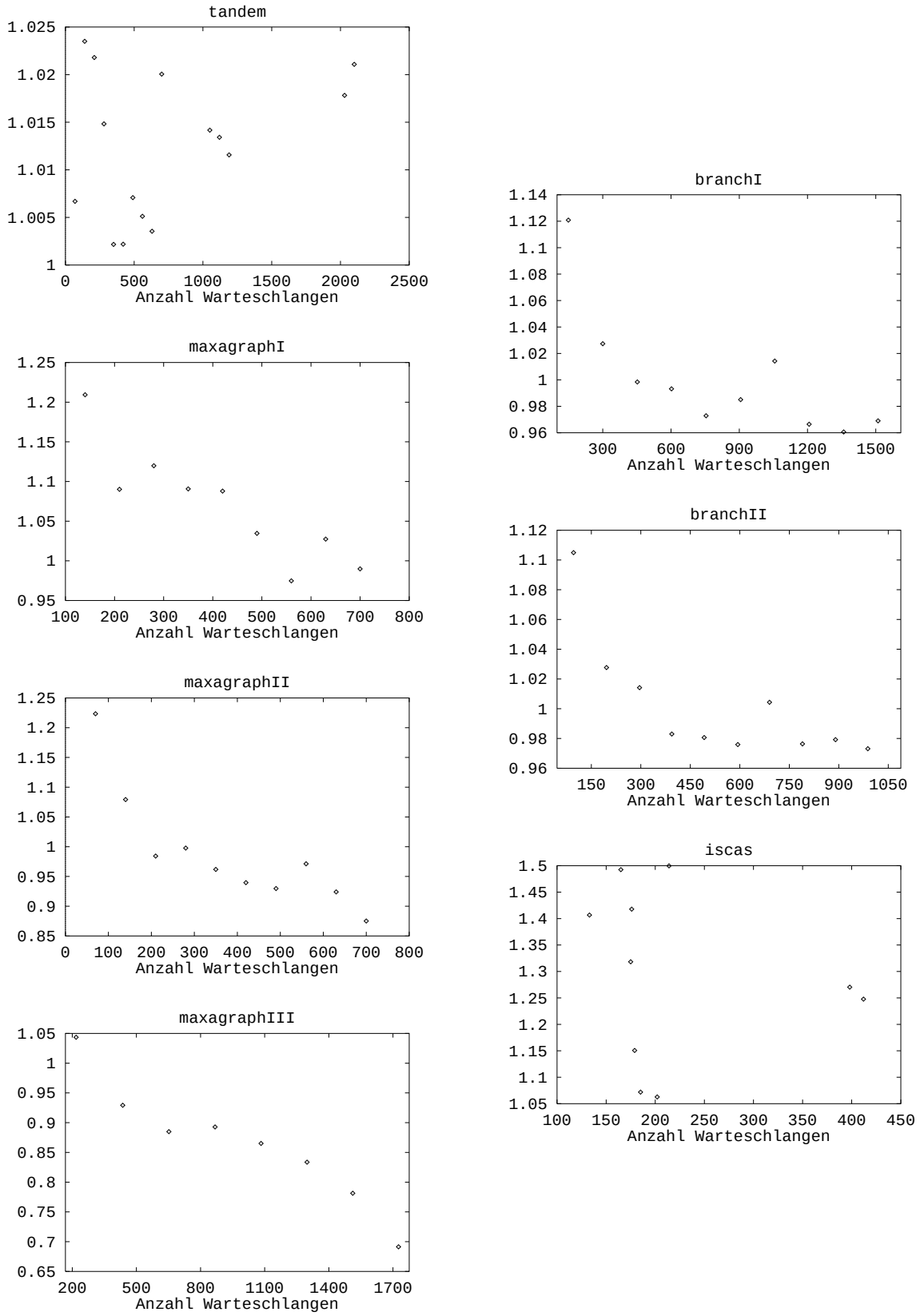


Abbildung 6.23: $\frac{t_{avoid-}}{t_{link}}$ für verschiedene Modelle

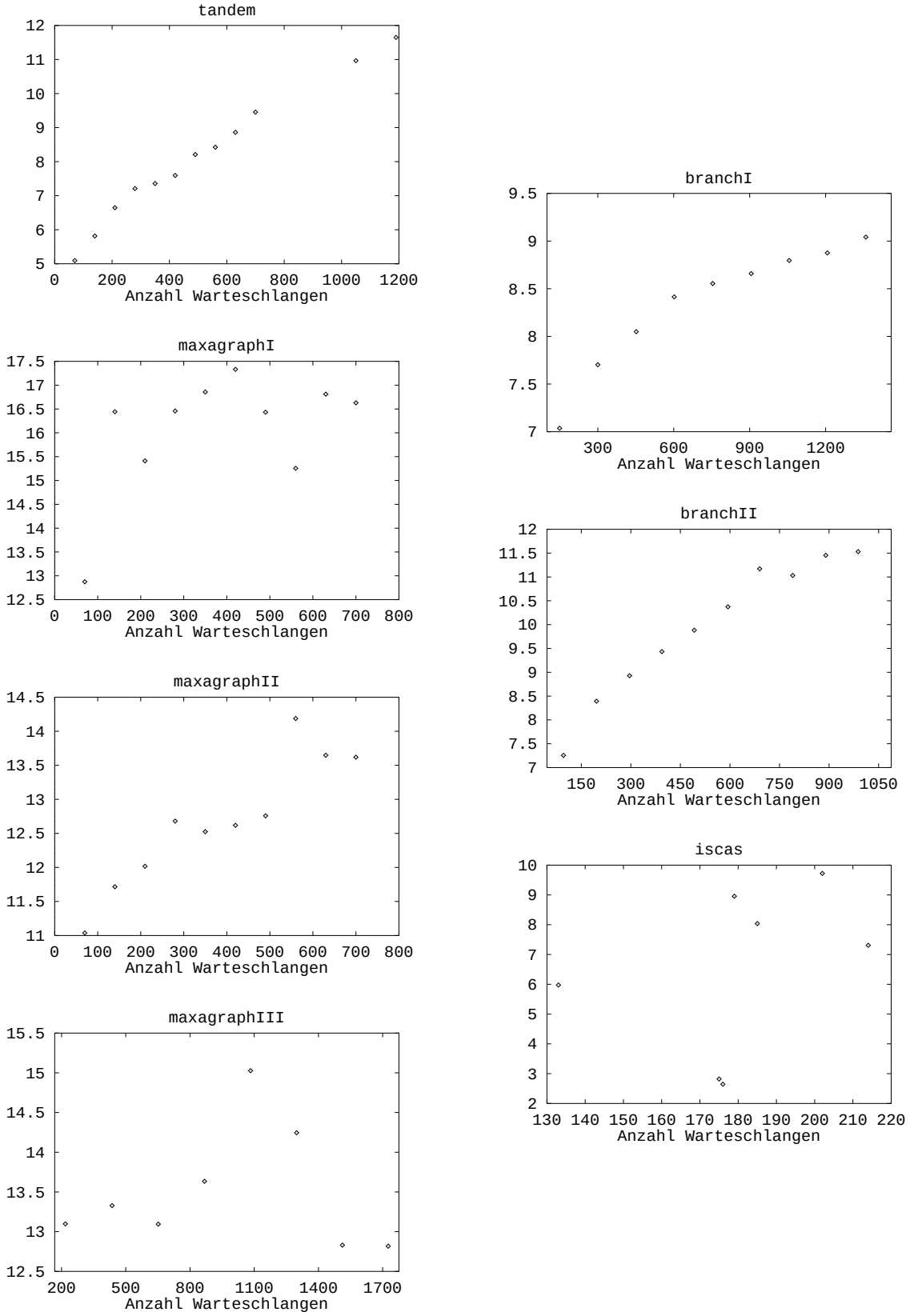


Abbildung 6.24: $\frac{t_{avoid}}{t_{link}}$ für verschiedene Modelle

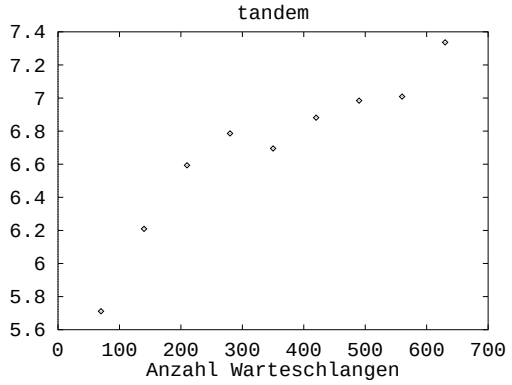


Abbildung 6.25: b_{link} für `tandem`

tionslauf ergaben, verglichen.

An und für sich hätten dabei die Beschleunigungen mit Hilfe der Laufzeit von `simulate` berechnet werden müssen. Da diese Werte aus den in 6.4 dargestellten Gründen dann aber nur unwesentlich über den Werte 1 hinausgekommen wären, hätte der Vergleich so gut wie keine Aussagekraft besessen. Daher wurde statt dessen die Laufzeit des parallelen, mit dem Linktime-Verfahren synchronisierten Simulators zugrunde gelegt, bei dem die Ausführung eines Ereignisses genauso aufwendig wie bei einem echt parallelen Simulationslauf ist. Da, wie in 6.5.2 bereits gezeigt wurde, das Linktime-Verfahren die Ausführung von Ereignissen nur unwesentlich verzögert, stellt der Vergleich der so errechneten Beschleunigung mit den Ergebnissen der Kritischen-Pfad-Analyse eine echte "Belastungsprobe" für die Kritische-Pfad-Analyse dar.

Gemessen wurden folgende Größen:

b_{kpa}	: Ergebnis der Kritischen-Pfad-Analyse
$b_{link} = \frac{t_{clu}}{t_{link}}$	: t_{clu} bzw. t_{link} die Laufzeit des parallelen, mit dem Linktime-Verfahren synchronisierten Simulators mit einem bzw. mehreren Simulatorknoten

Abb. 6.26 zeigt den Wert des Quotienten $\frac{b_{link}}{b_{kpa}}$ für verschiedene Modelle. Wie man sieht, hat b_{kpa} im Regelfall tatsächlich die Eigenschaft einer oberen Schranke, die, wie die Ergebnisse aus 6.5.2 vermuten ließen, vergleichsweise scharf ausfällt. Es gibt jedoch auch Ausnahmen, wie einige `tandem`- und `branch`-Modelle zeigen. Einen Hinweis auf die Ursache für diese "superkritische" Beschleunigung erhält man, wenn man sich die Werte von b_{kpa} für die `tandem`-Modelle anschaut (Abb. 6.25): Teilweise ist die Beschleunigung größer als 7 bei 7 verwendeten Prozessoren, also superlinear.

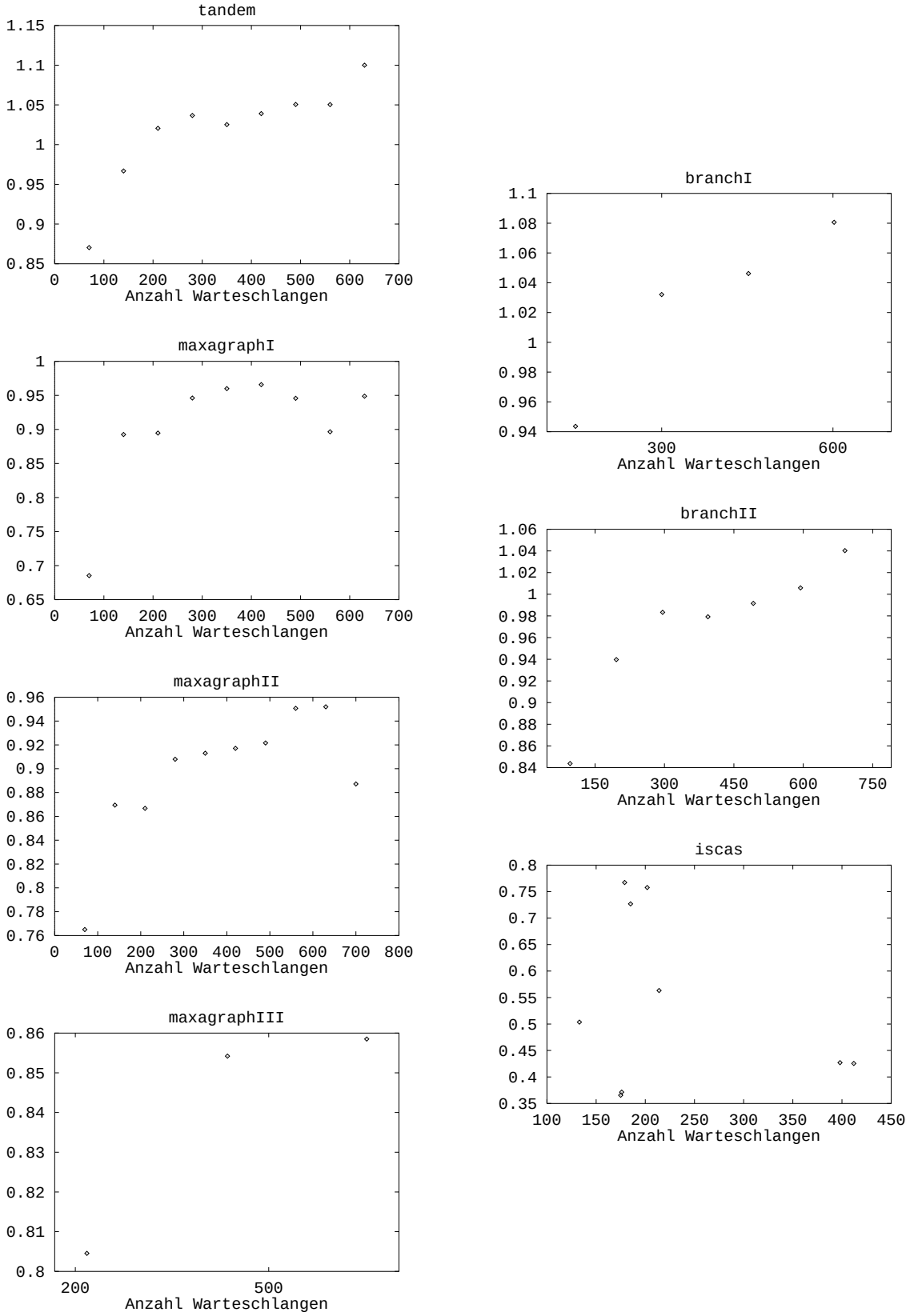


Abbildung 6.26: $\frac{b_{link}}{b_{kpa}}$ für verschiedene Modelle

Da sowohl für t_{clu} als auch für t_{link} derselbe Simulator mit demselben Synchronisationsverfahren verwendet wurde und dieses für t_{link} eher mehr Aufwand verursacht als für t_{clu}, läßt dies eigentlich nur den Schluß zu, daß sich die Ausführungszeiten von Ereignissen bei echt paralleler Simulation verkleinern, was auch die "superkritische" Beschleunigung bei den anderen Modellen erklären würde. Denkbare Gründe hierfür sind z. B. das in 6.1 beschriebene Verhalten der malloc-Funktion, seltenerer Cache-Miss bei Zugriffen des Prozessors auf den Hauptspeicher, etc. Da die Ausführungszeiten von Ereignissen bei echt parallelen Simulationsläufen aus den am Ende von 5.3.3 bereits dargelegten Gründen nur sehr schwer meßbar sind, konnte die genaue Ursache dieses Phänomens jedoch leider nicht näher untersucht werden.

Die Werte für b_{link} bei den `tandem`-Modellen demonstrieren im übrigen noch einmal eindrucksvoll die Effizienz eines in der Literatur immer wieder angewandten "Tricks", mit dessen Hilfe sich hervorragende Beschleunigungswerte beim Vergleich paralleler und sequentieller Simulationsläufe erzielen lassen. Er besteht darin, daß statt einem echten sequentiellem Simulator einfach ein paralleler Simulator verwendet wird, bei dem sämtliche Komponenten auf einem Rechnerknoten plaziert werden und die Laufzeit des damit durchgeführten "pseudoparallelen" Simulationslaufs (der häufig noch uneffizienter sein dürfte als der für t_{clu} gemessene Lauf mit einem einzigen Simulatorknoten) als Laufzeit des sequentiellen Simulators bei der Berechnung von Beschleunigungswerten benutzt wird.

Zusammenfassend kann man also sagen, daß die Ergebnisse der Kritischen-Pfad-Analyse hier tatsächlich obere Schranken für die erreichbaren Beschleunigungen liefern, jedoch naturgemäß nicht Phänomene, die auf verkürzten Ausführungszeiten von Ereignissen bei parallelen Simulationsläufen beruhen, berücksichtigen können. Weiterhin stellt sich natürlich auch die Frage, wie scharf die so ermittelten oberen Schranken sind, d. h. ob es beispielsweise möglich ist, mit den Ergebnissen der Kritischen-Pfad-Analyse auf die mit paralleler Simulation in der Praxis erzielbaren Beschleunigungen rückzuschließen. Die Antwort lautet im allgemeinen leider nein, da bei der Kritischen-Pfad-Analyse nicht alle in der Praxis auftretenden Probleme berücksichtigt werden. Dies zeigt unter anderem auch die Zusammenfassung der in dieser Arbeit durchgeführten Messungen, die im nun folgenden Kapitel vorgenommen wird.

Kapitel 7

Zusammenfassende Bewertung der Ergebnisse

Ausgehend von der am Anfang von Kapitel 6 aufgeworfenen Fragestellung "Welches sind die wesentlichen Probleme bei der Parallelisierung ereignisgesteuerter Simulationen?" wurden dort die Probleme in Einzelanalysen untersucht. In diesem Kapitel geht es nun darum, sie zu gewichten, um so Randprobleme von zentralen Problemen zu unterscheiden. Dazu werden in 7.1 die Analysen aus Kapitel 6 noch einmal zusammengefaßt und gegeneinander gestellt. Das sich dabei ergebende Gesamtbild läßt zwei Hauptprobleme klar erkennen. Dieses Ergebnis wird anschließend noch mit weiteren Analysen bestätigt, bei denen die Ausführung von Ereignissen künstlich verlangsamt wird (7.2) und parallele Simulationsläufe mit 32 Rechnerknoten (d. h. mit der maximal auf dem iPSC860 zur Verfügung stehenden Anzahl) durchgeführt werden (7.3).

7.1 Zusammenfassung der bisherigen Ergebnisse

Bei der Zusammenfassung der Ergebnisse aus Kapitel 6 stellt sich zunächst das Problem, daß die untersuchten Gründe für die schlechte Parallelisierbarkeit ereignisgesteuerter Simulationen zum Teil recht unterschiedlicher Natur sind. So führt zum Beispiel die Parallelisierung des Simulationsmodells (etwa durch das notwendige Aufspalten von Ereignissen) zu einer Erhöhung des Rechenaufwandes, während die Synchronisationsalgorithmen lediglich die Ausführung von Ereignissen verzögern, nicht aber den Rechenaufwand erhöhen. Es stellt sich daher die Frage nach einem geeigneten Vergleichskriterium für das "Gewicht" der verschiedenen Probleme.

Der weiter unten vorgenommene Vergleich basiert auf einer Folge von sieben (hypothetischen oder real durchgeführten) parallelen Simulationsläufen, bei denen von Folgeglied zu Folgeglied jeweils eines der untersuchten Probleme zusätzlich berücksichtigt wird. Für diese Simulationsläufe werden dann die sich gegenüber einem sequentiellen Simulationslauf ergebenden Beschleunigungen berechnet und miteinander verglichen. Je stärker zwei parallele Simulationsläufe sich in ihren Beschleunigungswerten unterscheiden, desto "gewichtiger" ist das neu hinzugekommene Problem. Abb. 7.1 beschreibt die Simulationsläufe mit den zugehörigen Beschleunigungswerten. Diese stellen insgesamt eine immer realistischer werdende Sicht auf einen parallelen Simulationslauf dar, wobei das Spektrum von "völlig utopisch" (Beschleunigung = Anzahl Simulatorknoten) bis "absolut realistisch" (tatsächliche Beschleunigung gegenüber ausoptimierter sequentieller Simulation) reicht. Entsprechend sollten die Beschleunigungswerte für jedes Simulationsmodell von 1 bis 7 monoton fallen, was aber aus den in Kapitel 6 bereits erläuterten Gründen in der Praxis nicht immer der Fall ist.

Abb. 7.2 zeigt die Maximal-, Minimal- und Mittelwerte für diese Beschleunigungen, berechnet über sämtliche Modelle der jeweiligen Modellreihe (soweit die entsprechenden Meßwerte verfügbar waren). Man erkennt dabei deutlich, daß zwei Problemursachen alle anderen deutlich dominieren: Die ungleiche Lastverteilung (Unterschied zwischen 1 und 2), die allerdings nicht bei allen Modellen auftritt, und die Parallelisierung des Simulatorcodes (Unterschied zwischen 5 und 6), die die Beschleunigung bei 7 (bzw. 6) Rechnerknoten auf Werte im Bereich 1 und darunter reduziert. Dies bestätigt noch einmal, was sich bei den Analysen in Kapitel 6 abzeichnete, nämlich daß Lastbalancierung und die feine Granularität der Anwendung die Haupthindernisse bei der effizienten Parallelisierung ereignisgesteuerter Simulationen darstellen.

Die Verbesserung der Lastbalancierung ist mittlerweile ein vielbeachtetes Forschungsgebiet innerhalb der parallelen Simulation (vgl. etwa [NL92, NL93, SB93]). Um Lösungen aufzuzeigen, mit denen sich die aus der Granularität der Anwendung ergebenden Probleme bewältigen lassen, werden im folgenden noch Messungen analysiert, bei denen einmal die Granularität der Anwendung künstlich durch eine Leerschleife während der Ausführung eines Ereignisses hochgesetzt wurde und zum anderen die Zahl der Simulatorknoten erhöht wurde, um so den (pro ausgeführtem Ereignis konstant bleibenden) Zusatzaufwand relativ zur erzielbaren Beschleunigung zu verkleinern. Die erstgenannte Methode stellt dabei natürlich nur insofern eine "Lösung" dar, als daß sie zeigt, daß bei Modellen mit höheren Ereignisausführungszeiten als den hier untersuchten gute Beschleunigungswerte möglich sind.

	Beschreibung	Zusätzlich zum vorhergehenden Simulationslauf berücksichtigtes Problem	Beschl.	Beschr. der Größen in
1	Optimale parallele Simulation, alle Simulatorknoten sind gleich stark und ständig ausgelastet, kein zusätzlicher Rechenaufwand gegenüber sequentieller Simulation vorhanden		n	6.3
2	Hypothetischer, paralleler Simulationslauf, bei dem jeder Simulatorknoten alle auszuführenden Ereignisse bereits beim Start der Simulation kennt, kein zusätzlicher Rechenaufwand gegenüber sequentieller Simulation vorhanden	Ungleiche Lastverteilung auf den Simulatorknoten	b_{last}	6.3
3	Hypothetischer, paralleler Simulationslauf, bei dem jeder Simulatorknoten ein Ereignis erst ausführen darf, nachdem es bei der Ausführung eines anderen Ereignisses erzeugt wurde, kein zusätzlicher Rechenaufwand gegenüber sequentieller Simulation vorhanden	Kausale Abhängigkeiten zwischen den Ereignissen	b_{kpa}	6.3
4	Real durchgeführter, paralleler Simulationslauf, bei dem jedes Ereignis so früh wie möglich ausgeführt wird (Orakeldateien), kein zusätzlicher Rechenaufwand bei der Ausführung von Ereignissen gegenüber sequentieller Simulation vorhanden	Verzögerung und zusätzlicher Aufwand für das Ein-, Auspacken und Verschicken von Ereignisnachrichten	$\frac{t_{clu}}{t_{ora}}$	6.4, 6.5.2
5	Real durchgeführter, paralleler Simulationslauf, synchronisiert mit dem Linktime-Verfahren, kein zusätzlicher Rechenaufwand bei der Ausführung von Ereignissen gegenüber sequentieller Simulation vorhanden	Verzögerung durch das Synchronisationsverfahren	$\frac{t_{clu}}{t_{link}}$	6.4, 6.5.2
6	Real durchgeführter, paralleler Simulationslauf, synchronisiert mit dem Linktime-Verfahren, Berücksichtigung des zusätzlichen Aufwandes für die Ausführung von Ereignissen gegenüber sequentieller Simulation	Parallelisierung des Simulatorcodes	$\frac{t_{sim}}{t_{link}}$	6.2, 6.5.2
7	Real durchgeführter, paralleler Simulationslauf, synchronisiert mit dem Linktime-Verfahren, Beschleunigung gegen optimierten sequentiellen Simulationslauf gemessen	Parallelisierung des Simulationsmodells	$\frac{t_{opt}}{t_{link}}$	6.2, 6.5.2

Abbildung 7.1: Simulationsläufe zur zusammenfassenden Bewertung der Analysen

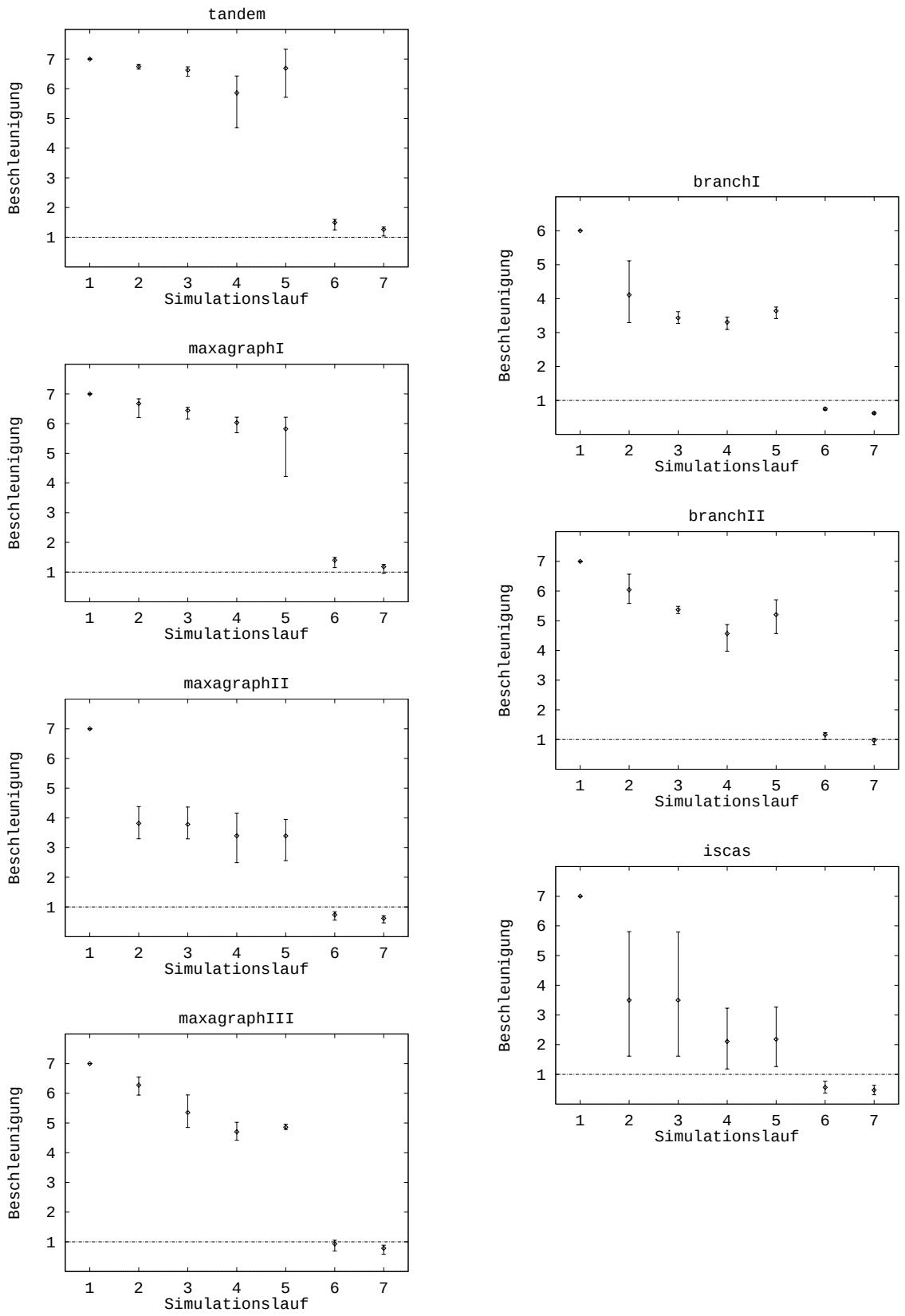


Abbildung 7.2: Zusammenfassung der Analysen und Meßergebnisse

7.2 Simulation mit künstlich verlangsamten Ereignissen

Bei den in diesem Abschnitt durchgeführten Analysen wurde die Ausführung von Ereignissen künstlich verzögert, indem in den C-Code sowohl der sequentiellen als auch der parallelen Simulatoren beim Start der Ereignisausführung eine Schleife der Form

```
for (i = 1; i++; i <= MAXLOOPS)
```

eingefügt wurde. Eine Alternative hätte darin bestanden, mit Hilfe der von MMK zur Verfügung gestellten `waitask`-Funktion den Simulatorprozeß für eine gewisse Zeit zu passivieren, jedoch wurde durch das "Busy-Waiting" der Schleife die CPU des jeweiligen Simulatorknotens bewußt blockiert, da dies auch während der Ausführungszeit der Ereignisse der Fall ist und es darum ging, diese Zeit künstlich zu verlängern.

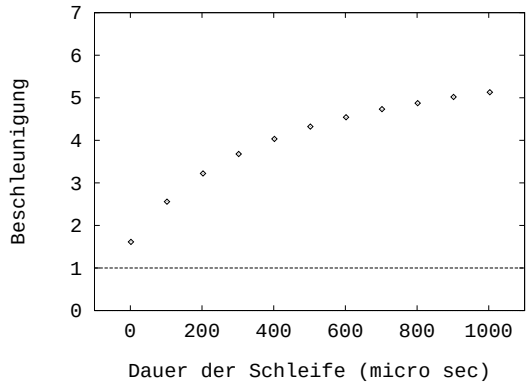


Abbildung 7.3: Zusammenhang zwischen Dauer der Verzögerungsschleife und Beschleunigung bei einem `tandem`-Modell mit 700 Warteschlangen auf 7 Knoten

Direkte Messungen auf dem iPSC860 ergaben, daß erwartungsgemäß die durch die Schleife bewirkte Verzögerung linear vom Wert von `MAXLOOPS` abhing. Ebenfalls erwartungsgemäß ergaben dieselben Werte für `MAXLOOPS` sowohl beim sequentiellen als auch beim verteilten Simulator dieselben Verzögerungszeiten, so daß es insgesamt möglich war, durch geeignete Wahl der Werte von `MAXLOOPS` Verzögerungszeiten einzustellen, die in etwa Vielfachen von $100\mu s$ entsprachen, wodurch die mittlere Ausführungszeit der Ereignisse vervielfacht wurde (vgl. 6.1). Für die Laufzeitmessungen der sequentiellen Läufe wurde der sequentielle Simulator `simulate` verwendet, da dieser dasselbe Simulationsmodell wie der parallele Simulator benutzte und es daher bei ihm Sinn machte, dieselbe Verzögerung wie beim parallelen Simulator

zu verwenden. Für den parallelen Simulator wurde Linktime als Synchronisationsverfahren gewählt.

Abb. 7.3 zeigt die bei einem `tandem`-Modell mit 700 Warteschlangen erzielten Beschleunigungen auf 7 Rechnerknoten. Wie man deutlich sieht, reichen bereits Verzögerungen von $200\mu s$ aus, um die Beschleunigung auf akzeptable Werte (3.5 mit 7 Simulatorknoten) zu bringen. Die Kurve flacht dann mit steigenden Verzögerungszeiten immer mehr ab, wobei sie sich einem asymptotischen Wert nähert, der vermutlich lediglich durch die unterschiedlichen Anzahlen ausgeführter Ereignisse der verschiedenen Simulatorknoten bestimmt wird.

Insgesamt unterstützen dabei auch diese Messungen die Vermutung, daß mit steigender Granularität der Anwendung alle bisher untersuchten Probleme bis auf die Lastbalancierung an Bedeutung stark verlieren[1].

7.3 Simulation mit 31 Simulatorknoten

Die in diesem Abschnitt analysierten Messungen wurden mit 32 Rechnerknoten (31 Simulatorknoten und 1 Knoten für die Ablaufsteuerung) durchgeführt, d. h. mit der Gesamtzahl der Rechnerknoten, die auf dem für diese Arbeit benutzten iPSC860-System zur Verfügung standen. Bei der Auswahl der Warteschlangennetze für die Simulationsläufe schieden eine Reihe von Modellen von vornherein aus, da auf dem entsprechenden Rechnerknoten nicht genügend Platz für das Parsen ihrer Beschreibung war. Weiterhin wurde, da es im wesentlichen darum ging, zu zeigen, daß mit 31 Simulatorknoten der zusätzliche Aufwand für die parallele Simulation durch die erzielte Beschleunigung mehr als ausgeglichen wird, auf Simulationsmodelle verzichtet, bei denen die Lastbalacierung stark den Parallelismus dämpft. Übrig blieben noch einige `tandem`- und `maxagraphI`-Modelle, die analog zu den in 6.1 beschriebenen Modellen für 31 Simulatorknoten konstruiert wurden.

Abb. 7.4 zeigt die Ergebnisse der Kritischen-Pfad-Analyse und die durch parallele Simulation (synchronisiert mit dem Linktime-Verfahren) erzielte Beschleunigung, wobei letztere gegen einen Simulationslauf mit `optsim` gemessen wurden. Wie man sieht, ist der laut Kritischer-Pfad-Analyse vorhandene Parallelismus bei allen Modellen recht hoch (27-28 bei den `tandem`- und 24-25 bei den `maxagraphI`-Modellen). Trotzdem fällt die tatsächlich erzielte Beschleunigung im Vergleich zur Zahl der

[1] Dies hat natürlich zur Folge, daß bei ineffizient programmierten Anwendungen hohe Beschleunigungswerte erheblich leichter zu erzielen sind als bei effizient programmierten. Man kann vermuten, daß dies eine weitere Quelle für einige in der Literatur genannte extrem gute Beschleunigungswerte ist.

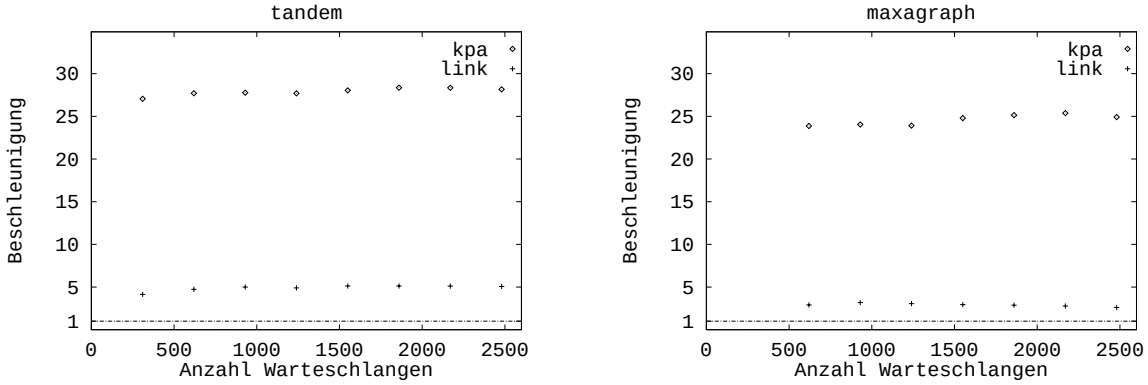


Abbildung 7.4: Beschleunigungsmessungen mit 31 Knoten

Simulatorknoten eher bescheiden aus (ca. 5 bei den `tandem`- und ca. 3 bei den `maxagraphI`-Modellen). Der Hauptgrund liegt wohl in dem in 6.4 bereits analysierten zusätzlichen Rechenaufwand für die Parallelisierung des Simulatorcodes. Geht man dabei grob geschätzt vom vierfachen der Rechenzeit der sequentiellen Simulation aus, so reduziert allein dies die potentielle Beschleunigung auf 25 % des Wertes der Kritischen-Pfad-Analyse. In diesem Licht betrachtet stellen die tatsächlich erzielten Beschleunigungen bereits recht gute Werte dar.

Zusammenfassend läßt sich sagen, daß sich bei den untersuchten parallelen Simulationsläufen die erzielten Beschleunigungen durch den Einsatz einer größeren Zahl von Simulatorknoten deutlich steigern ließen, wobei sie aber verglichen mit der Zahl der Simulatorknoten aus den bereits analysierten Gründen eher moderat ausfielen.

Kapitel 8

Ein Kapitel für sich: Optimistische Synchronisationsverfahren

8.1 Einleitung

Bedingt durch die hohe Komplexität der Algorithmen und die schwer überschaubaren Folgen der verfrühten Ausführung von Ereignissen, gelten optimistische Synchronisationsverfahren als schwer realisierbar, insbesondere was die Effizienz der Verfahren betrifft[1]. Bei der hier vorgestellten Realisierung für den parallelen Simulator des DISQUE-Testbetts bestätigte sich dies einmal mehr, so daß im Rahmen dieser Arbeit das Verfahren weder vollständig ausoptimiert noch detailliert untersucht werden konnte. Dennoch sollen hier einige Ergebnisse vorgestellt werden, um einerseits die speziellen Probleme, die bei der Realisierung des Verfahrens im DISQUE-Testbett auftraten, darzustellen, und um andererseits einen ersten Vergleich des Laufzeitverhaltens von parallelen Simulationsläufen mit optimistischen und konservativen Synchronisationsverfahren vorzunehmen.

[1] Jack Briner schrieb beispielsweise im Anhang seiner Dissertation, in der es um die Realisierung eines optimistisch synchronisierten, parallelen Simulators ging: "Developing the parallel simlator took about three times the effort and time expected." [Bri90]

8.2 Aspekte der Realisierung optimistischer Synchronisationsverfahren für den parallelen Simulator des DISQUE-Testbetts

Ein erstes, eher technisches Hindernis, das bei der Realisierung eines optimistischen Simulationsverfahrens auftrat, war das in 5.2.2 beschriebene Filterkonzept. Rückblickend läßt sich feststellen, daß dieses Konzept, obwohl es zunächst nur für die Realisierung konservativer Verfahren in [RD89] entwickelt wurde, sich prinzipiell auch für optimistische Verfahren eignet. Jedoch müssen dabei die besonderen Belange optimistischer Verfahren bei der Definition der Schnittstellen der Filter berücksichtigt werden, was bei der Konzeption des parallelen Simulators leider nicht geschah. (Beispielsweise wurde der dynamisch angeforderte Speicher für ein Ereignis grundsätzlich nach dessen Ausführung freigegeben. Bei optimistischen Verfahren ist dies jedoch nicht korrekt, da das Ereignis unter Umständen nach einem Rollback erneut ausgeführt werden muß.) Da die Neukonzeption der Filterschnittstellen jedoch umfangreiche Änderungen am parallelen Simulator erfordert hätte, wurde darauf verzichtet und statt dessen einige Teile des Simulatorcodes außerhalb der Filter für das optimistische Verfahren geändert.

Ein weiteres Problem, das wohl bei vielen Realisierungen optimistischer Synchronisationsverfahren auftritt und sich vor allem negativ auf das Laufzeitverhalten des Simulators auswirkte, war die regelmäßige Erzeugung von Kopien der komplexen Datenstrukturen der von den Simulatorknoten verwalteten Teilzustände des Simulationsmodells. Diese bestehen im wesentlichen aus den Zustandsbeschreibungen der Warteschlangen und ihrer Klassen (Statistiken, Seed-Werte für die Zufallszahlengeneratoren, etc.) und der an diesen Warteschlangen wartenden Kunden (angeforderte Bedienzeit, Reihenfolge im Warteraum, etc.). Das Sichern eines solchen Zustands erfordert neben dem vollständigen Durchlaufen von Strukturen wie Arrays oder verketteten Listen auch das Anfordern von Speicher für eine große Zahl kleinerer Objekte. Wegen des vergleichsweise hohen Aufwands für die `malloc`-Funktion (vgl. 6.1) erwies es sich als dringend notwendig, für diese Objekte Freispeicherlisten einzuführen, was den Aufwand für die Zustandssicherung auf ca. 30% des ursprünglichen Wertes reduzierte. Trotzdem blieb, wie später noch näher ausgeführt wird, das Kopieren der Datenstrukturen eine rechenzeitintensive Operation.

Schließlich zeigte sich, daß optimistisch synchronisierte, parallele Simulationen doch erheblich speicheraufwendiger sind als konservativ synchronisierte. Dies führte bei den durchgeführten Experimenten bei einigen Simulationsläufen zu temporären Speicherengpässen auf einzelnen Simulatorknoten, die daraufhin nicht mehr weiterarbeiten konnten. Hier machte sich das Fehlen eines verteilten Speichermanagements (wie

z. B. die in 3.3 bereits erwähnten Verfahren in [Gaf88, Jef90, Lin92a]) bemerkbar. Da dies nur mit erheblichem Aufwand zu realisieren gewesen wäre, wurde statt dessen die Möglichkeit geschaffen, dem Synchronisationsverfahren ein Zeitfenster vorzugeben, d. h. die Anzahl der Datenkopien, die ein Simulator anlegen darf, zu begrenzen. Dabei führt ein Simulator, der diese Grenze überschritten hat, solange keine Ereignisse mehr aus, bis er sie durch den Anstieg der GVT (Global Virtual Time, vgl. 3.3) und der damit verbundenen fossil collection wieder unterschreitet. Bei den weiter unten beschriebenen Messungen wurden dann diese Zeitfenster so gewählt, daß keine Speicherengpässe mehr auftraten. Dies führte natürlich zu einem deutlichen Anstieg der Laufzeit der Simulation, erwies sich jedoch für die unten beschriebenen Messungen als nicht allzu problematisch, da die meisten Simulationsläufe völlig ohne Zeitfenster auskamen.

Zur Bestimmung einer unteren Schranke für die GVT wurde ein Algorithmus verwendet, der auf dem Schnappschuß-Algorithmus von Chandy und Lamport [CL85] beruht. Bei ihm übernimmt ein Simulatorknoten die Rolle des Initiators, der in regelmäßigen Abständen durch das Verschicken einer Nachricht an alle anderen Simulatorknoten eine GVT-Bestimmung auslöst. Diese schicken bei Erhalt der Nachricht über alle Kanäle (d. h. über alle Verbindungen, die sie für das Versenden von Ereignis- und Antinachrichten an andere Simulatorknoten benutzen) eine sogenannte Markernachricht. Weiterhin bestimmen sie den Zeitstempel des vordersten Ereignisses ihrer Ereignisliste. Anschließend führen sie fortlaufend das Minimum über die Zeitstempel aller eintreffenden Ereignis- und Antinachrichten, auf deren Kanal noch keine Markernachricht eingetroffen ist. Nach dem Eintreffen der letzten Markernachricht senden sie das Minimum der beiden Werte an den Initiator. Dieselben Aktionen können dabei bereits durch das Eintreffen einer Markernachricht ausgelöst werden, in diesem Fall wird die später vom Initiator eintreffende Nachricht ignoriert. Der Initiator bildet dann das Minimum aller eintreffenden Werte und schickt dieses als neue GVT-Approximation an die anderen Simulatorknoten. Dieses Verfahren setzt FIFO-Kanäle zwischen den einzelnen Simulatorknoten voraus, damit die Markernachrichten nicht andere Nachrichten überholen können. Abschließend sei noch angemerkt, daß es effektivere Verfahren zur GVT-Bestimmung gibt (wie etwa die in [Mat93] beschriebenen), der zusätzliche Nachrichten- und Rechenaufwand jedoch bereits bei dem realisierten Verfahren sehr gering ist und die GVT ohnehin nur zur fossil collection verwendet wird. Somit ist der Einfluß des GVT-Algorithmus auf die Gesamtlaufzeit der Simulation eher gering zu bewerten.

Da im HELIOS-Programmiermodell nur Punkt-zu-Punkt-Verbindungen (Pipes) existierten, wurde das Aussenden der Initiator-Nachrichten, das Einsammeln der Minima sowie das Aussenden der neuen GVT-Approximation mit Hilfe von ringförmig über Pipes verbundenen Prozessen auf den einzelnen Simulatorknoten realisiert,

wobei auf dem Pipe-Ring lediglich ein Token kreist, das die entsprechenden Informationen enthält.

8.3 Untersuchung des Laufzeitverhaltens von Simulationsläufen mit optimistischen Synchronisationsverfahren

Bedingt durch die Kürze der zur Verfügung stehenden Zeit und die Speicherplatzprobleme bei der Simulation größerer Modelle wurden die in diesem Abschnitt beschriebenen Messungen lediglich mit den Modellen der `tandem`-Reihe durchgeführt. Dabei ging es im wesentlichen um zwei Fragen:

- Welche Länge hat das optimale Checkpointintervall[2]?
- Wie und wie stark unterscheidet sich die Laufzeit eines optimistisch synchronisierten Simulationslaufs mit optimal gewähltem Checkpointintervall von der eines konservativ synchronisierten Simulationslaufs?

Bei den im folgenden beschriebenen Messungen wurden sämtliche Simulationsläufe mit 7 Simulatorknoten durchgeführt, wobei beim Versenden der Antinachrichten die Lazy-Cancellation-Strategie angewendet wurde.

8.3.1 Bestimmung des optimalen Checkpointintervalls

Abb. 8.1 zeigt die Laufzeiten der optimistisch synchronisierten Simulationsläufe in Abhängigkeit von der verwendeten Checkpointintervallänge und der Größe des simulierten `tandem`-Modells. Speziell markiert wurden dabei die Laufzeiten, bei denen wegen Speichermangel ein Zeitfenster verwendet werden mußte, sowie das Minimum aller zu jeweils einem Modell gehörenden Laufzeiten. Der fehlende Wert für die Checkpointintervallänge 1 bei 700 Warteschlangen erklärt sich daraus, daß der zugehörige Simulationslauf nach mehreren Stunden noch nicht beendet war und daher abgebrochen wurde.

[2] Wie in der Literatur allgemein üblich, werden im folgenden die regelmäßig angelegten Kopien der Datenstrukturen des Simulators als Checkpoints und die Zahl der zwischen dem Anlegen zweier solcher Kopien ausgeführten Ereignisse als Länge des Checkpointintervalls bezeichnet.

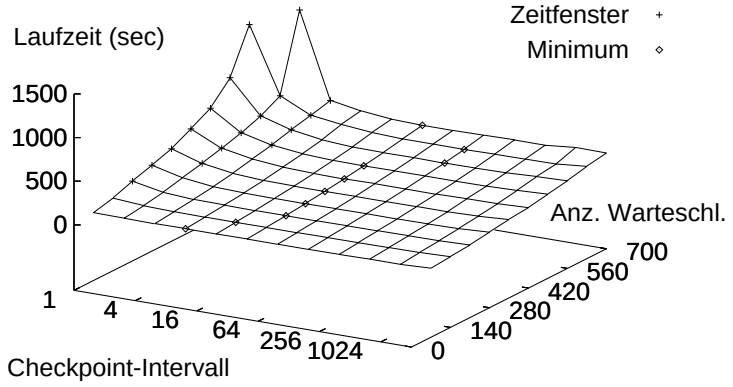


Abbildung 8.1: Laufzeit optimistisch synchronisierter Simulationsläufe für verschiedene `tandem`-Modelle

Wie man deutlich erkennt, ist die optimale Checkpointintervallänge deutlich größer als 1, teilweise liegt sie sogar um den Wert 128. Der explosionsartige Anstieg der Laufzeiten bei kleiner werdenden Checkpointintervalllängen ist dabei zwar sicherlich auf die immer schmaler werdenden Zeitfenster zurückzuführen, jedoch gibt die Form der Fläche der Laufzeiten ohne Zeitfenster Anlaß zu der Vermutung, daß sich auch ohne Zeitfenster die optimale Checkpointintervallänge nicht ändern würde. Dies ist insofern bemerkenswert, als in der Literatur lange Zeit davon ausgegangen wurde, daß es normalerweise am effektivsten ist, nach jedem ausgeführten Ereignis einen Checkpoint zu erzeugen. Beispielsweise war, wie S. Bellenot in [Bel92] ausführt, in dem in 3.5.2 beschriebenen Time Warp Operating System das Verwenden von Checkpointintervallen vorgesehen, wurde jedoch später fallengelassen, da Testläufe ergaben, daß das Erzeugen von Checkpoints nach jedem ausgeführten Ereignis am effektivsten war. Daß dies nicht immer der Fall sein muß, ist eine Erkenntnis der jüngeren Zeit. In [Bel92] findet sich dafür eine interessante Erklärung: Es wird dort zwischen gesättigten, parallelen Simulationsläufen unterschieden, bei denen die Simulatorknoten fast ständig mit der Ausführung von Ereignissen beschäftigt sind, und ungesättigten, bei denen die Simulatorknoten nur teilweise mit der Ereignisausführung ausgelastet sind. Bei den ungesättigten Simulationsläufen fällt die Erzeugung von Checkpoints vielfach in Zeiten, in denen der entsprechende Simulatorknoten ohnehin beschäftigungslos ist, was den gesamten Simulationslauf nicht nennenswert verzögert. Bei den gesättigten Simulationsläufen ist dies in der Regel nicht der Fall. Da in beiden Fällen die Coasting-Forward-Phase (d. h. das erneute Ausführen der Ereignisse, die zeitlich zwischen Checkpoint und Straggler liegen, vgl.

3.3) für zusätzlichen Rechenaufwand sorgt, kann dies dazu führen, daß es bei ungesättigten Simulationsläufen im Gegensatz zu gesättigten am günstigsten ist, nach jedem ausgeführten Ereignis einen Checkpoint zu erzeugen. Da man auf Grund der guten Werte der Kritischen-Pfad-Analyse (vgl. 6.3) und des Vergleichs der konservativ synchronisierten, parallelen Simulationsläufe auf einem und mehreren Knoten (Simulationslauf Nr. 5 in 7.1) wohl davon ausgehen kann, daß die untersuchten Simulationsläufe gesättigt sind, stehen die hier gemessenen Werte in vollem Einklang mit dieser Theorie.

Bei diesen Überlegungen wurde natürlich implizit vorausgesetzt, daß das Erzeugen von Checkpoints einen nicht vernachlässigbaren Rechenaufwand verursacht. Um dies für die untersuchten Simulationsläufe zu bestätigen, wurden diese mit optimaler Checkpointintervallänge wiederholt und dabei die zeitliche Länge der Checkpointerzeugung gemessen: Abb. 8.2 zeigt Mittelwert und Standardabweichung (dargestellt

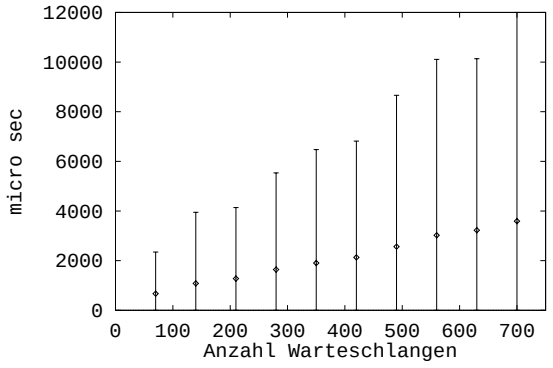


Abbildung 8.2: Aufwand für das Erzeugen von Checkpoints für verschiedene `tandem`-Modelle

als Intervall der Länge der doppelten Standardabweichung um den Mittelwert) der zeitlichen Dauer der Checkpointerzeugung, gemessen über alle 7 Simulatorknoten. Wie man sieht, steigen die Mittelwerte linear mit der Modellgröße bis auf Werte, die in etwa dem 40-fachen der mittleren Ausführungszeit eines Ereignisses entsprechen ($100\mu s$, vgl. 6.1). Berücksichtigt man noch, daß die einzelnen Meßwerte sehr stark um den Mittelwert streuen, so kann man davon ausgehen, daß das Erzeugen von Checkpoints eine aufwendige Operation darstellt. Dies gilt zunächst nur für die durch das hier verwendete Simulationsmodell vorgegebenen Datenstrukturen. Man kann aber wohl davon ausgehen, daß bei ähnlichen Simulationsmodellen, wie sie etwa bei der Simulation von Materialflußsystemen oder Rechnernetzen auftreten, das Erzeugen von Checkpoints ähnlich aufwendig ist. Darüber hinaus stellt sich natürlich auch die Frage, in wieweit es möglich ist, die Datenstrukturen so zu

organisieren, daß sie einfacher (z. B. ohne Berücksichtigung der genauen Struktur als zusammenhängender Speicherblock) kopiert werden können. Dieser Aspekt, der im Prinzip wie bei den Untersuchungen in 6.2 eine Änderung des Simulationsmodells für die speziellen Anforderungen der parallelen Simulation darstellt, konnte im Rahmen dieser Arbeit leider nicht mehr untersucht werden.

Eine weitere Eigenschaft der hier untersuchten Simulationsläufe, die die Verwendung von Checkpointintervallen bei den hier untersuchten Simulationsläufen begünstigt, ist die geringe Anzahl auftretender Rollbacks. Abb. 8.3 zeigt die Gesamtzahl der auf allen Simulatorknoten aufgetretenen Rollbacks in Abhängigkeit von Checkpointin-

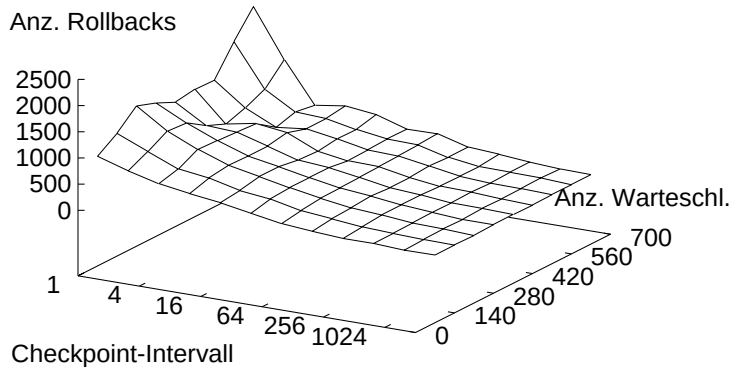


Abbildung 8.3: Gesamtzahl Rollbacks für verschiedene `tandem`-Modelle

tervallänge und Modellgröße. Vergleicht man die dort auftretenden Werte mit der Zahl der im entsprechenden Simulationslauf ausgeführten Ereignisse (vgl. 6.1), so sieht man, daß Rollbacks eher selten auftreten und so die Kosten für die Coasting-Forward-Phase wohl nicht sehr stark ins Gewicht fallen. Verstärkt wird dieser Effekt noch dadurch, daß die Zahl der Rollbacks mit steigender Checkpointintervallänge sinkt. Dies ist wohl darauf zurückzuführen, daß mit steigender Checkpointintervallänge die Coasting-Forward-Phase länger dauert und somit sowohl das Ausführen als auch das Erzeugen und Versenden von Ereignissen verzögert wird, was die Häufigkeit von Stragglern reduziert.

Zusammenfassend läßt sich also feststellen, daß bei den hier untersuchten Modellen bedingt durch die hohe Auslastung der Simulatorknoten, den hohen Aufwand für das Erzeugen von Checkpoints und die geringe Anzahl von Rollbacks, sich die Laufzeit des parallelen Simulationslaufs durch die Verringerung der Zahl der Checkpoints

deutlich verbessern läßt, was, wie bereits weiter oben ausgeführt wurde, wohl auf dem "gesättigten" Charakter des parallelen Simulationslaufs beruht.

8.3.2 Laufzeitvergleich optimistisch und konservativ synchronisierter Simulationsläufe

Als Grundlage des hier durchgeführten Vergleichs wurden folgende Laufzeiten benutzt:

t_{link} : Laufzeit des mit dem Linktime-Verfahren synchronisierten, parallelen Simulationslaufs

t_{tw} : Laufzeit des optimistisch synchronisierten, parallelen Simulationslaufs

Abb. 8.4 zeigt den Wert des Quotienten $\frac{t_{tw}}{t_{link}}$ für die `tandem`-Modellreihe. Wie man

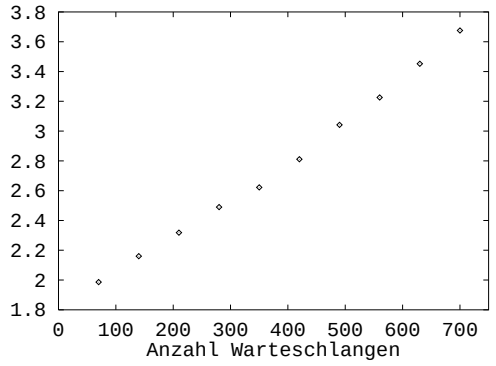


Abbildung 8.4: $\frac{t_{tw}}{t_{link}}$ für verschiedene `tandem`-Modelle

sieht, läuft die optimistisch synchronisierte Simulation deutlich langsamer als die mit dem Linktime-Verfahren synchronisierte, wobei der Wert des Quotienten sogar linear mit der Modellgröße steigt. Hierbei muß man sich vor Augen halten, daß, wie in 6.5.2 gezeigt wurde, das Linktime-Verfahren bei den `tandem`-Modellen die Ereignisausführung so gut wie überhaupt nicht behindert, jedes Ereignis also frühestmöglich ausgeführt wird (d. h. sobald es auf dem entsprechenden Simulatorknoten erzeugt bzw. eingetroffen und an der Reihe ist). Somit bringt ein optimistisches Synchronisationsverfahren bei diesen Modellen fast keine Vorteile, da bei ihnen im Regelfall die Ereignisse auch nicht früher korrekt ausgeführt werden können. Andererseits benötigen sie natürlich zusätzlich Rechenzeit für das Erzeugen von Checkpoints und

für das wiederholte Ausführen von bereits korrekt ausgeführten Ereignissen während der Coasting-Forward-Phase, was wohl den Unterschied zwischen t_{tw} und t_{link} ausmacht.

Insgesamt ergaben sich also beim Vergleich optimistisch und konservativ synchronisierter Simulationsläufe bei der optimistischen Synchronisation erheblich schlechtere Laufzeiten, die wohl im wesentlichen darauf zurückzuführen sind, daß bei den verwendeten Simulationsmodellen die konservativen Verfahren bereits annähernd optimal funktionieren. Die Messungen stellen damit den "worst case" für optimistisch synchronisierte Simulationen dar und liefern einen Anhaltspunkt, welchen Zusatzaufwand optimistische Synchronisation gegenüber konservativer Synchronisation verursacht. Dieser Zusatzaufwand "amortisiert" sich natürlich bei vielen Modellen, z. B. wenn der Graph der Simulatorknoten Zyklen enthält und der Lookahead (vgl. 3.2) des Simulationsmodells nur gering ist.

Kapitel 9

Zusammenfassung und Ausblick

Die Entwicklung der praktischen Informatik ist bis heute durch ständig steigenden Umfang der eingesetzten Software und immer höhere Anforderungen der Anwender gekennzeichnet. Die Leistungssteigerung der eingesetzten Rechnersysteme durch die Entwicklung immer leistungsfähigerer Prozessoren kann damit zwar im Augenblick immer noch Schritt halten, jedoch gibt es technologisch bedingte Grenzen, die dabei nicht überschritten werden können. Damit gewinnt die im Prinzip seit langem bestehende Idee, die Ausführung von Anwendungen durch Parallelisierung, d. h. durch Zerlegen des zu lösenden Problems in Teilprobleme, die von mehreren Prozessoren getrennt bearbeitet werden können, zu beschleunigen, immer mehr an Bedeutung. Da in den letzten Jahren in zunehmendem Maß Mehrprozessorsysteme kommerziell verfügbar wurden, wird mittlerweile verstärkt nach Anwendungen gesucht, bei denen Parallelisierung in der Praxis auch wirklich zu beschleunigter Ausführung der Anwendung führt.

Einen Beitrag hierzu wollte auch die vorliegende Arbeit leisten. Untersucht wurde eine Klasse von Anwendungen – ereignisgesteuerte Simulation – die sowohl durch hohen Rechenzeitbedarf als auch durch große Praxisrelevanz gekennzeichnet ist und somit ein interessantes Versuchsfeld für den Einsatz von Parallelrechnern darstellt. Zunächst wurde dazu der Begriff des sequentiellen und des verteilten Simulationsmodells exakt definiert, um die grundlegenden Begriffe der ereignisgesteuerten Simulation sowie die durch die Parallelisierung an das Simulationsmodell gestellten Anforderungen exakt beschreiben zu können. Darauf aufbauend wurden die bisher existierenden Ansätze bei der parallelen, ereignisgesteuerten Simulation systematisch dargestellt.

Den Hauptteil dieser Arbeit stellte die experimentell-analytische Untersuchung ei-

ner speziellen Klasse von Simulationsmodellen, den sogenannten Warteschlangennetzen, dar. Dazu wurde zunächst in Anlehnung an RESQ2 von IBM die Sprache LAVENDER zur Beschreibung von Warteschlangennetzen entwickelt. Darauf aufbauend wurde ein umfangreiches Testbett (DISQUE) für die parallele Simulation von Warteschlangennetzen realisiert, in dem neben verschiedenen sequentiellen und parallelen Simulatoren für in LAVENDER beschriebene Warteschlangennetze auch Werkzeuge zur Analyse und Animation paralleler Simulationsläufe realisiert wurden. Mit Hilfe dieses Testbetts wurde dann der Frage nachgegangen, ob es möglich ist, durch Parallelisierung die Ausführung ereignisgesteuerter Simulationen nennenswert zu beschleunigen und welche Gründe dem gegebenenfalls entgegenstehen. Die mit Hilfe dieses Testbetts durchgeführten Beschleunigungsmessungen ergaben nur sehr geringe Beschleunigungswerte für die untersuchten Modelle; teilweise liefen die parallelen Simulationsläufe sogar länger als die sequentiellen.

Diese Erkenntnisse stehen im grundsätzlichen Einklang mit anderen veröffentlichten Ergebnissen; es wurden daraufhin die Gründe für die im allgemeinen schlechten Beschleunigungswerte untersucht. Hierbei wurde ein Analyseverfahren entwickelt, das es ermöglichte, die verschiedenen Ursachen zu gewichten, um so "relevante" von "nebensächlichen" Gründen zu unterscheiden. Das Verfahren beruht auf einer Folge von parallelen Simulationsläufen, die teilweise real durchgeführt werden, teilweise nur hypothetischer Natur sind (d. h. ihre Laufzeit wird mit Hilfe von anderweitig ermittelten Realzeitinformationen wie etwa den Ausführungszeiten von Ereignissen berechnet statt gemessen). Je zwei in der Folge aufeinanderfolgende Simulationen unterscheiden sich im wesentlichen darin, daß bei einem von beiden ein für das Erzielen von Beschleunigungen ungünstiger Aspekt zusätzlich berücksichtigt wird. Somit handelt es sich insgesamt um eine Folge von immer realistischer werdenden Simulationsläufen, bei der, wenn man sie mit demselben Simulationsmodell ausführt, die zugehörigen Beschleunigungswerte im allgemeinen monoton fallen. Die Differenz in den Beschleunigungswerten zweier aufeinanderfolgender Simulationsläufe liefert dann ein Maß für das "Gewicht" des zusätzlich berücksichtigten Aspekts.

Diese Analysemethode wurde auf eine Reihe sehr unterschiedlicher Simulationsmodelle angewendet. Dabei kristallisierten sich sehr deutlich zwei Hauptprobleme heraus: Die Lastbalancierung (d. h. die gleichmäßige Verteilung der auszuführenden Ereignisse auf die einzelnen Simulatoren des parallelen Simulators) und der zusätzlich zur eigentlichen Simulation anfallende Aufwand für Synchronisation und Kommunikation. Das erstere Problem tritt häufig bei der Programmierung paralleler Anwendungen auf, die Anpassung existierender Lösungen und das Entwickeln spezieller Lösungen für die parallele, ereignisgesteuerte Simulation stellt im Augenblick ein vielbeachtetes Forschungsgebiet dar, so daß, auch wenn es eine generelle Lösung dieses Problems wohl nicht geben wird, wohl doch in naher Zukunft Lösungen für

speziellen Anwendungen zu erwarten sind. Das zweite Problem resultiert aus der feinen Granularität der Anwendung und ist wohl im allgemeinen nur schwer lösbar: Die Ausführung eines Ereignisses benötigt derartig wenig Rechenzeit, daß bei paralleler Simulation für die regelmäßig davor oder danach auszuführenden Synchronisations- und Kommunikations-Aktionen nur wenig Zeit zur Verfügung steht, falls der dadurch entstehende Mehraufwand nicht ins uferlose wachsen soll. Da die Ausführungszeiten von Ereignissen durch das Simulationsmodell fest vorgegeben sind, läßt sich an dieser Tatsache im Prinzip nichts ändern.

Hierüber können auch gewisse, in der Literatur gelegentlich auftauchende sehr gute Beschleunigungswerte nicht hinwegtäuschen. Ein Nebenprodukt der in dieser Arbeit durchgeführten Untersuchungen war die Demonstration gewisser fragwürdiger Methoden, solche Werte zu erzielen. Die fälschlicherweise oft angewandte Praktik, einen parallelen Simulationslauf, bei dem alle Komponenten eines parallelen Simulators auf einem einzelnen Rechnerknoten plaziert werden, als sequentiellen Simulationslauf bei der Berechnung von Beschleunigungen zu verwenden, führte beispielsweise bei einigen der untersuchten Modelle zu superlinearen Beschleunigungswerten!

Das Gebiet der parallelen, ereignisgesteuerten Simulation ist jetzt über 10 Jahre alt. Wenn ein Gebiet der praktischen Informatik dieses Alter erreicht hat, ohne, wie im vorliegenden Fall, in nennenswertem Umfang Eingang in kommerziell verfügbare Produkte gefunden zu haben, stellt sich natürlich die Frage nach den Ursachen dafür. Einen gewichtigen Grund liefert die vorliegende Arbeit: Parallele, ereignisgesteuerte Simulatoren stellen komplexe Anwendungen dar, deren Realisierung mit großem Aufwand verbunden ist. Erzielen lassen sich damit - wenn überhaupt - nur sehr mäßige Beschleunigungen gegenüber sequentieller, ereignisgesteuerter Simulation. Da dieses Problem, wie bereits erwähnt, im wesentlichen auf den kurzen Ausführungszeiten der Ereignisse beruht, handelt es sich dabei um modellinhärente Schwierigkeiten, die sich nicht durch effizientere Algorithmen etc. beseitigen lassen. Somit wird immer deutlicher (und die vorliegende Arbeit versteht sich unter anderem auch als Beitrag zu diesem Erkenntnisprozeß), daß sich parallele, ereignisgesteuerte Simulation wohl nicht auf breiter Front durchsetzen wird.

Dies heißt natürlich nicht, daß es generell keine Einsatzgebiete dafür gibt. Beispielsweise wurde das Projekt, in dem das Timewarp Operating System erstellt wurde, ursprünglich von der DARPA ins Leben gerufen, um Simulatoren, die auf verschiedenen, teilweise in verschiedenen Städten stehenden Rechnern installiert waren, miteinander zu koppeln. Eine weitere, bisher noch wenig untersuchte Anwendung von paralleler, ereignisgesteuerter Simulation stellt die Vergrößerung des zur Verfügung stehenden Hauptspeichers auf lokal vernetzten Workstations dar. So kommt es bei der Simulation großer VLSI-Schaltungen beispielsweise häufiger vor, daß der auf einer einzelnen Workstation zur Verfügung stehende reale Hauptspeicher bei wei-

tem nicht ausreicht und die bei der Verwendung von virtuellem Speicher notwendig werdenden Dateizugriffe die Laufzeit der Simulation erheblich in die Höhe treiben. Hier besteht eine gute Möglichkeit, durch den Einsatz paralleler, ereignisgesteuerter Simulation den auf mehreren Workstations vorhandenen realen Hauptspeicher gleichzeitig zu nutzen. Die dadurch überflüssig werdenden Dateizugriffe sollten den zusätzlich anfallenden Synchronisations- und Kommunikationsaufwand bei paralleler Simulation kompensieren.

Abschließend sei noch auf die Frage eingegangen, welche zukünftigen Arbeiten sich noch an die vorliegende anschließend könnten: Interessant wäre zunächst einmal die Weiterverfolgung der in dieser Arbeit durchgeführten Untersuchung optimistischer Synchronisationsverfahren, insbesondere mit Modellen, bei denen keine effiziente, konservative Synchronisation möglich ist. Weiterhin wäre es, besonders im Hinblick auf die bereits erwähnten zukünftigen Anwendungen paralleler, ereignisgesteuerter Simulationen, interessant, die in dieser Arbeit durchgeführten Analysen mit der auf dem Workstation-Netz laufenden Version von DISQUE im Hinblick auf die Frage zu wiederholen, ob sich die Gewichtung der analysierten Probleme gegenüber den mit dem iPSC860 durchgeführten Analysen verschiebt. Um die Allgemeingültigkeit der Resultate der mit dem DISQUE-Testbett durchgeführten Analysen zu untermauern, wäre es außerdem noch wünschenswert, die im Rahmen dieser Arbeit entwickelte Analysemethode auf andere parallele Simulatoren anzuwenden, was bei den meisten Simulatoren ohne weiteres möglich sein müßte. Schließlich sollte sich das Analyseverfahren recht gut auf parallele Anwendungen übertragen lassen, bei denen in Objekten ununterbrechbare Aktionen durch Nachrichten angestoßen werden (beispielsweise bei Anwendungen, die mit Actor- oder parallelen, objektorientierten Sprachen realisiert wurden). Das so übertragene Analyseverfahren könnte dann zu interessanten neuen Einblicken in das Verhalten der entsprechenden Anwendung führen.

Literaturverzeichnis

[BBK89] F. Brglez, D. Bryan, K. Kozminski. Combinational Profiles of Sequential Benchmark Circuits. In *Proc. IEEE International Symposium on Circuits and Systems*, S. 1929–1934, 1989.

[BBLT90] T. Bemmerl, A. Bode, T. Ludwig, S. Tritscher. MMK-Multiprocessor Multitasking Kernel. SFB-Bericht 342/26/90 A, Technische Universität München, 1990.

[Bel90] S. Bellenot. Global Virtual Time Algorithms. In *Proceedings of the SCS Multiconference on Distributed Simulation*, S. 122–127, 1990.

[Bel92] S. Bellenot. State Skipping Performance with the Time Warp Operating System. In *Proceedings of the SCS Western Simulation MultiConference on Parallel and Distributed Simulation*, S. 53–64, 1992.

[BJ85] O. Berry, D. R. Jefferson. Critical Path Analysis of Distributed Simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, S. 57–60, 1985.

[Bol89] G. Bolch. *Leistungsbewertung von Rechensystemen mittels analytischer Warteschlangenmodelle.* B. G. Teubner, Stuttgart, 1989.

[Bri90] J. V. Briner. *Parallel Mixed-Level Simulation of Digital Circuits Using Virtual Time.* PhD thesis, Duke University, 1990.

[Bro82] M. Broy. *A Theory for Nondeterminism, Parallelism, Communication and Concurrency.* Habilitationsschrift, Technische Universität München, 1982.

[Bro88] R. Brown. Calendar Queues: A Fast O(1) Priority Queue Implemetation for the Simulation Event Set Problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[Bry79] R. E. Bryant. Simulation on a Distributed System. In *1st International Conference on Distributed Computer Systems*, S. 544–552, 1979.

[Buz71] J. P. Buzen. *Queueing Network Models of Multiprogramming.* PhD thesis, Harvard University, 1971.

[CL85] K. M. Chandy, L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[CM81] K. M. Chandy, J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, 24(11):198–205, 1981.

[CSR93] K. Chung, J. Sang, V. Rego. A Performance Comparison of Event Calendar Algorithms: an Empirical Approach. *Software - Practice and Experience*, 23(10):1107–1138, 1993.

[CW67] A. B. Carroll, R. T. Wetherald. Application of Parallel Processing to Numerical Weather Prediction. *Journal of the ACM*, 14(3):591–614, 1967.

[CW91] P. Chawla, P. A. Wilsey. Synchronizing Distributed VHDL Simulation. Technischer Bericht TR 131-4-91, Dept. of Electrical & Computer Engineering, University of Cincinnati, 1991.

[Dun90] R. Duncan. A Survey of Parallel Computer Architectures. *Computer*, 23(2):5–16, 1990.

[EDLP+89] M. Ebling, M. Di Loreto, M. Presley, F. Wieland, D. R Jefferson. An Ant Foraging Model Implemented on the Time Warp Operating System. In *Proceedings of the SCS Mulitconference on Distributed Simulation*, S. 21–26, 1989.

[Eve79] S. Even. *Graph Algorithms.* Pitman Publishing Ltd., London, 1979.

[Fuj88] R. M. Fujimoto. Lookahead in Parallel Discrete Event Simulation. In *Proceedings of the International Conference on Parallel Processing*, 1988.

[Fuj90] R. M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[Gaf88] A. Gafni. Rollback Mechanisms for Optimistic Distributed Simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, S. 61–67, 1988.

[HE91] M. T. Heath, J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8(5):29–39, 1991.

[JBH+85] D. R. Jefferson, B. Beckman, S. Hughes, E. Levy, T. Litwin, J. Spagnuolo, J. Vavrus, F. Wieland, B. Zimmerman. Implementation of Time Warp on the Caltech Hypercube. In *Proceedings of the SCS Multiconference on Distributed Simulation*, S. 70–75, 1985.

[JCRB89] D. W. Jones, C. Chou, D. Renk, S. C. Bruell. Experience with Concurrent Simulation. In *Proceedings of the Winter Simulation Conference*, S. 756–764, 1989.

[Jef85] D. R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[Jef90] D. R. Jefferson. Virtual time II: The Cancelback Protocol for Storage Management in Time Warp. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, S. 75–90, 1990.

[JS82] D. R. Jefferson, H. Sowizral. Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control. RAND Note N-1906AF, The Rand Corporation, 1982.

[KL70] B. W. Kernighan, S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49:291–307, 1970.

[Kle78] L. Kleinrock. *Queueing Systems*, Band II. John Wiley, New York, 1978.

[LDF91] G. Lomow, S. R. Das, R. M. Fujimoto. User Cancellation of Events in Time Warp. In *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, S. 55–62, 1991.

[Lin92a] Y. Lin. Memory Management Algorithms for Optimistic Parallel Simulation. In *Proceedings of the SCS Western Simulation MultiConference on Parallel and Distributed Simulation*, S. 43–52, 1992.

[Lin92b] Y. Lin. Parallelism Analyzer for Parallel Discrete Event Simulation. *ACM Transactions on Modeling and Computer Simulation*, 2(3):239–264, 1992.

[LL90] Y. Lin, E. Lazowska. Reducing the State Saving Overhead for Time Warp Parallel Simulation. In *Proceedings of the International Conference on Parallel Processing*, S. 201–209, 1990.

[LP91] Y. Lin, B. R. Preiss. Optimal Memory Management for Time Warp Parallel Simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(4):283–307, 1991.

[Ltd90] Perihelion Software Ltd. *The Helios Operating System.* Prentice Hall, 1990.

[LW93] J. I. Leivent, R. J. Watro. Mathematical Foundations for Time Warp Systems. *ACM Transactions on Programming Languages and Systems*, 15(5):771–794, 1993.

[Mar94] M. Marks. Implemetierung graphischer Darstellungsmöglichkeiten für YES und Bewertung des Logiksimulators DVSIM mit YES. In Vorbereitung: Diplomarbeit, Universität des Saarlandes, 1994.

[Mat89] F. Mattern. *Verteilte Basisalgorithmen.* Informatik-Fachberichte 226. Springer-Verlag, 1989.

[Mat93] F. Mattern. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. *Journal of Parallel and Distributed Computing*, 18:423–434, 1993.

[Meh91] H. Mehl. Breaking Ties Deterministically in Distributed Simulation Schemes. Technischer Bericht 217/91, Universität Kaiserslautern, FB Informatik, 1991.

[Meh94] H. Mehl. *Methoden verteilter Simulation.* Vieweg-Verlag, 1994.

[Mei91] G. Meister. Verteilte Simulation von Warteschlangennetzen auf Transputern. Projektarbeit, Universität Kaiserslautern, 1991.

[Mei92] G. Meister. Implementierung der Time-Warp-Strategie für den verteilten Warteschlangennetz-Simulator DISQUE. Diplomarbeit, Universität Kaiserslautern, 1992.

[Mil89] D. L. Mills. *RFC 1129: Internet Time Synchronisation: The Network Time Protocol.* Network Information Center, SRI International, 1989.

[Mis86] J. Misra. Distributed Discrete-Event Simulation. *ACM Computing Surveys*, 18(1):39–65, 1986.

[MM89] F. Mattern, H. Mehl. Diskrete Simulation - Prinzipien und Probleme der Effizienzsteigerung durch Parallelisierung. *Informatik Spektrum*, 12(4):198–210, 1989.

[MMST91] F. Mattern, H. Mehl, A. Schoone, G. Tel. Global Virtual Time Approximation with Distributed Termination Detection Algorithms. Technischer Bericht RUU-CS-91-32, Universität Utrecht, 1991.

[Moh90] B. Mohr. Performance Evaluation of Parallel Programs in Parallel and Distributed Systems. In *Proceedings of the VAPP/CONPAR 90*, LNCS 457. Springer-Verlag, 1990.

[Mül93] A. Müller. Implementierung eines "Nullmessages on demand" Protokolles in DISQUE. Fortgeschrittenen-Praktikum, Universität des Saarlandes, 1993.

[Mye92] W. Myers. High-performance computing is 'window into the future', says President's science advisor. *Computer*, 25(1):87–90, 1992.

[Nic88] D. M. Nicol. High Performance Parallelized Discrete Event Simulation of Stochastic Queueing Networks. In *Proceedings of the Winter Simulation Conference*, S. 306–314, 1988.

[NL92] B. Nandy, W. M. Loucks. An Algorithm for Partitioning and Mapping Conservative Parallel Simulation onto Multicomputers. In *Proceedings of the SCS Western Simulation MultiConference on Parallel and Distributed Simulation*, S. 139–146, 1992.

[NL93] B. Nandy, W. M. Loucks. On a Parallel Partitioning Technique for Use with Conservative Parallel Simulation. In *Proceedings of the Workshop on Parallel and Distributed Simulation*, S. 43–51, 1993.

[PEWJ89] M. Presley, M. Ebling, F. Wieland, D. R Jefferson. Benchmarking the Time Warp operating system with a computer network simulation. In *Proceedings of the SCS Mulitconference on Distributed Simulation*, S. 8–13, 1989.

[PM86] K. A. Pickar, J. D. Meindl. Integrated Circuit Technologies of the Future. *Proceedings of the IEEE, Special Issue*, 74(12), 1986.

[PM88] S. K. Park, K. W. Miller. Random Number Generators: Good Ones are Hard to Find. *Communications of the ACM*, 31(10):1192–1201, 1988.

[Poh91] W. Pohlmann. A Fixed Point Approach to Parallel Discrete Event Simulation. *Acta Informatica*, 28:611–629, 1991.

[PR85] J. Postel, J. Reynolds. *RFC 959: File Transfer Protocol (FTP)*. Network Information Center, SRI International, 1985.

[RD89] P. F. Reynolds, Ph. M. Dickens. SPECTRUM: A Parallel Simulation Testbed. In *Proceedings of the Hypercube Conference*, 1989.

[Ric91] J. Richter. DISQUE: Ein verteilter Simulator für Warteschlangennetze auf Transputern. In R. Grebe, M. Baumann (Hrsg.), *Parallele Datenverarbeitung mit dem Transputer (3. Transputer-Anwender-Treffen)*, S. 345–352. Springer-Verlag, 1991.

[Rös93] K. Rössig. Eine Übersicht über aktuelle Ansätze zur verteilten optimistischen Simulation. Technischer Bericht AIS-11, Universität Oldenburg, FB Informatik, Arbeitsgruppe Informationssysteme, 1993.

[Run88] D. Runo. Das Graphpartitionierungsproblem und seine Lösungsmethoden. Diplomarbeit, GMD Bonn, 1988.

[RW89] R. Righter, J. C. Walrand. Distributed Simulation of Discrete Event Systems. *Proceedings of the IEEE*, 77(1):99–113, 1989.

[Sam85] B. Samadi. *Distributed Simulation, Algorithms and Performance Analysis*. PhD thesis, University of California, 1985.

[SB93] Ch. Sporrer, H. Bauer. Corolla Partitioning for Distributed Logic Simulation of VLSI-Circuits. In *Proceedings of the Workshop on Parallel and Distributed Simulation*, S. 85–92, 1993.

[SC91] H. S. Stone, J. Cocke. Computer Architecture in the 1990s. *Computer*, 24(9):30–38, 1991.

[Sch91] W. Schmitz. Verteilte Simulation von Warteschlangennetzen. Projektarbeit, Universität Kaiserslautern, 1991.

[Sch93] M. Schösser. Oracle-Lag für DISQUE. Fortgeschrittenen-Praktikum, Universität des Saarlandes, 1993.

[SCS89] T. K. Som, B. A. Cota, R. G. Sargent. On Analyzing Events to Estimate the Possible Speedup of Parallel Discrete Event Simulation. In *Proceedings of the Winter Simulation Conference*, S. 729–737, 1989.

[SF87] S. M. Swope, R. M. Fujimoto. Optimal Performance of Distributed Simulation Programs. In *Proceedings of the Winter Simulation Conference*, S. 612–617, 1987.

[SMK84] C. H. Sauer, E. A. MacNair, J. F. Kurose. The Research Queueing Package Version 2: Introduction and Examples. Technischer Bericht RA 138, IBM Thomas J. Watson Research Center, 1984.

[Sou92] L. P. Soulé. Parallel Logic Simulation: An Evaluation of Centralized-Time and Distributed-Time Algorithms. Technischer Bericht CSL-TR-92-527, Stanford Computer System Laboratory, 1992.

[SS90] J. Sang, M. Sang. Untersuchung von Algorithmen zur verteilten ereignisgesteuerten Simulation. Diplomarbeit, Universität Dortmund, 1990.

[Stu93] B. Sturm. YES: Ein Werkzeug zur Analyse verteilter Simulationsalgorithmen. Diplomarbeit, Universität Kaiserslautern, 1993.

[Su89] W.-K. Su. Reactive-Process Programming and Distributed Discrete Event-Simulation. Technischer Bericht Caltech-CS-TR-89-11, Computer Science Department, California Institute of Technology, 1989.

[WHF+89] F. Wieland, L. Hawley, A. Feinberg, M. Di Loreto, L. Blume, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, D. R Jefferson. Distributed Combat Simulation and Time Warp: The Model and its Performance. In *Proceedings of the SCS Mulitconference on Distributed Simulation*, S. 14–20, 1989.

[Win93] S. Winterstein. Vektormethode und Echo-Algorithmus für DISQUE. Fortgeschrittenen-Praktikum, Universität des Saarlandes, 1993.

[WRJ93] F. Wieland, P. Reiher, D. Jefferson. Experiences in Parallel Performance Measurement: The Speedup Bias. unveröffentlicht, 1993.

[Zei84] B. Zeigler. *Multifaceted Modelling and Discrete Event Simulation*. Academic Press, London, 1984.