Simulation pulsverarbeitender neuronaler Netze

Eine ereignisgetriebene und verteilte Simulation pulsverarbeitender neuronaler Netze

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Cyprian Graßmann

aus

Köln

Bonn, im April 2003

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

1.Referent: Prof. Dr. Joachim K. Anlauf
2.Referent: Prof. Dr. Christoph Strelen
Tag der Promotion:

Kapitel 1 - Einleitung 7

1.1. Aufgabenstellung 7

1.2. Kapitelübersicht 8

Kapitel 2 - Pulsverarbeitende neuronale Netze 11

2.1. Das Gehirn als Modell für neuronale Netzwerke 12

2.1.1. Die Struktur des Gehirns 12

2.1.2. Vernetzung 15

2.1.3. Das Neuron und Neuronenarten 16

2.1.4. Die Synapse – Signalfortleitung 19

2.1.5. Aktivität 21

2.1.6. Zusammenfassung der Eckdaten und tabellarische Übersicht 21

2.2. Modelle pulsverarbeitender Neuronen 23

2.2.1. Hodgkin-Huxley-Modell 24

2.2.2. Abschnitts-Modelle – compartment models 26

2.2.3. Integrate-and-Fire-Modell 27

2.2.4. Spike-Response-Modell 28

2.2.5. Eckhorn-Modell 30

2.2.6. Verlustlose Integrate-and-Fire-Modelle 32

2.2.7. Stochastische Modelle 33

2.3. Modelle der Informationscodierung und -verarbeitung 34

2.3.1. Modelle der Codierung 34

2.3.2. Lernen in pulsverarbeitenden neuronalen Netzen 35

Kapitel 3 - Simulation pulsverarbeitender neuronaler Netze 36

3.1. Grundlegende Verfahren 37

3.1.1. Sequentielle Simulation 37

3.1.2. Verteilte Simulation 47

3.2. Ereignisgetriebene Simulation pulsverarbeitender neuronaler Netze 52

3.2.1. Leistungscharakteristik zeitgetriebener und ereignisgetriebener Simulation 52

3.2.2. Zeitliche Auflösung und numerische Genauigkeit 53

3.2.3. Verarbeitung kontinuierlicher Signale 57

3.2.4. Stochastische Ereignisse – spontane Aktivität 57

3.2.5. Behandlung von Verzögerungen im Netzwerk 58

3.3. Optimierung der ereignisgetriebenen Simulation 59

3.3.1. Zentrale Simulationsschleife 59

3.3.2. Neuronenmodelle 63

3.3.3. Verteilte Simulation 66

3.4. Nachrichtensysteme zur Verteilung der Simulation 69

3.4.1. TCP und UDP 71

3.4.2. MPI 71

3.4.3. PVM 72

3.4.4. CORBA 73

3.4.5. COM/DCOM 74

3.4.6. Leistungsvergleich der Nachrichtensysteme 74

3.4.7. Auswahl des Nachrichtensystems 80

3.5. Verwandte Arbeiten und Simulationssysteme 81

3.5.1. Zeitscheibensimulation 81

3.5.2. Spezialisierte Zeitscheibensimulation 81

3.5.3. Ereignisgetriebene Simulationsverfahren 82

Kapitel 4 - Das SPIKELAB Simulationssystem 84

4.1. Das Gesamtsystem im Überblick – Hard- und Software 84

4.1.1. Topologie – Graphische Oberfläche - 3D Visualisierung 84

4.1.2. Automatische Partitionierung 87

4.1.3. Ereignisgetriebene und verteilte Simulation 87

4.1.4. Analyse von Simulationsergebnissen 88

4.1.5. RACER – Hardwarebeschleunigung 88

4.2. Softwarearchitektur 88

4.2.1. Die Simulatorschnittstelle 90

4.2.2. Subsimulator 90

4.2.3. Subsimulator in einer verteilten Simulation 94

4.2.4. Koordination der verteilten Simulation 94

4.2.5. Behandlung "zufälliger" Ereignisse 94

4.2.6. Realisierung von Lernvorgängen 95

4.2.7. Hardwareanbindung 96

4.3. Leistungsbewertung der Software 97

4.3.1. Vergleich mit einem Zeitscheibensimulator 97

4.3.2. Leistungscharakteristik der ereignisgetriebenen Simulation 104

4.3.3. Leistung der verteilten Simulation 114

Kapitel 5 - Hardware für die Simulation pulsverarbeitender neuronaler Netze 121

5.1. Beschleunigung der ereignisgetriebenen und verteilten Simulation pulsverarbeitender neuronaler Netze 121

5.1.1. Beschleunigung der Kommunikation 121

5.1.2. Beschleunigung der Ereignisliste 122
5.1.3. Beschleunigung der Neuronenmodelle 122

5.2. Neurocomputer und Simulationsbeschleuniger 122
5.2.1. Neurocomputer - digitale Hardwarebeschleunigung für PVNN 123

Kapitel 6 - RACER - Hardware zur Beschleunigung der Simulation und Integration von pulserzeugender Hardware 128

6.1. Aufbau der Evaluierungsplattform (PLD-Karte) 128
6.1.1. Architektur 128
6.1.2. Die Realisierung 129
6.1.3. Programmierumgebung 132

6.2. Leistungsbewertung der Hardware 136
6.2.1. Ereignisliste 136
6.2.2. Neuronenmodelle 137

6.3. Ergebnisse und Folgerungen 139

Kapitel 7 - Zusammenfassung und Ausblick 141

Kapitel 8 - Anhang 143

8.1. Installation des Softwarepakets SPIKELAB 143
8.1.1. Softwarepaket 143
8.1.2. Umgebungsvariablen 143
8.1.3. Übersetzung 143
8.1.4. RSH Deamon 143
8.1.5. CORBA 144

8.2. Installation des RACER Hardware-Treibers unter NT4 145

8.3. Handhabung der RACER-Hardware 146

Kapitel 9 - Referenzen 148

Kapitel 10 - SPIKELAB Bedienung und Programmierung 157

10.1. Graphische Oberfläche 157
10.1.1. Start des Programms 157

10.2. Netzwerkgeneratoren 160

10.3. Simulation 162

10.4. Kommandozeilenparameter für den Programmstart 163

10.5. Betrieb über die Konsole – Fileformat 164

10.5.1. Partitionsdatei 164
10.5.2. Gewichtsdatei 165
10.5.3. Log-Datei 166
10.5.4. Hostdatei 167

10.6. Programmierung eines Neuronenmodells 168
10.6.1. Klassenstruktur 168
10.6.2. Namenskonventionen 168
10.6.3. ParameterManager und ParameterSet 168
10.6.4. Schritte zur Implementierung 168
10.6.5. Logischer Prozess (MeinModellName.cpp/h) 169
10.6.6. ParameterManager (PM_MeinModellName.cpp/h) 171
10.6.7. GUI-Klassen 175

10.7. Programmierung eines Netzwerkgenerators 176
10.7.1. Namenskonventionen 176
10.7.2. Schnittstelle eines Netzwerkgenerators 177
10.7.3. Implementierung eines Netzwerkgenerators 177

Kapitel 11 - Abbildungsverzeichnis 179

Kapitel 12 - Tabellenverzeichnis 183

Kapitel 13 - Glossar 184

Kapitel 14 - Eidesstattliche Erklärung 187

Kapitel 15 - Zusammenfassung 188

Kapitel 1 - Einleitung

1.1. Aufgabenstellung

In den letzten Jahren wurden verstärkt zeitliche Aspekte bei der Signalverarbeitung durch pulsverarbeitende neuronale Netze (PVNN) erforscht. Die Beschäftigung mit PVNN ist durch neurobiologische Untersuchungen motiviert worden, wie [HodHux1952], [HubWie1959], [HubWie1962] und [HubWie1977]. Es folgten Arbeiten, die sich der Organisation solcher Netzwerke widmen, wie zum Beispiel [Malsburg1981] und [Malsburg1987] sowie deren technischen Realisierungen, wie [CanEck1990], [BeeJanEck1990], [Eckmiller1991] und [Eckmiller1993]. Die Artikelsammlungen [RieWarRuy1997] und [MaaBis1998] geben einen aktuellen Einblick in die theoretischen Überlegungen und zu den technischen Umsetzungen von PVNN.

Die Arbeiten [Maass1995], [Maass1996] zeigten auf, dass die Berechnungskomplexität pulsverarbeitender neuronaler Netzwerke höher ist als die von Netzwerken, welche mit einer Ratencodierung arbeiten. Die technische Verwertung dieser höheren Berechnungskomplexität ist die Triebfeder vieler Arbeiten, die sich mit PVNN beschäftigen.

Deshalb befassen sich nun auch zunehmend wissenschaftliche Arbeiten mit Lernverfahren für PVNN, weil sich Lernverfahren für ratencodierte neuronale Netzwerke, wie zum Beispiel *back propagation,* nicht auf PVNN anwenden lassen. Vielmehr werden Lernverfahren gesucht, die der zeitabhängigen Informationsverarbeitung in PVNN Rechung tragen.

Die Verarbeitung von Informationen findet in PVNN insofern zeitabhängig statt, als die Ausgaben von Neuronen nicht zu festen Zeitintervallen berechnet und ausgewertet werden, sondern entweder die absoluten Zeitpunkte, zu denen Neuronen einen Puls aussenden, oder die Relation zwischen solchen Zeitpunkten ausgewertet werden. So wird zum Beispiel bei der „*time to first spike"*-Codierung die Zeit bis zum ersten Puls nach einer Erregung eines PVNN als Informationsmaß herangezogen. In anderen Fällen wird zum Beispiel die Zahl und die räumliche Anordnung synchron pulsender Neuronen eines PVNN ausgewertet.

In einer Simulation von PVNN müssen daher die Pulszeitpunkte und die sie beeinflussenden Größen, wie zum Beispiel Verzögerungen und Anstiegszeiten, möglichst exakt repräsentiert und mit einer hohen Genauigkeit berechnet werden. Dem steht üblicherweise der Wunsch nach einer schnellen Simulation größerer PVNN entgegen.

Es gab bis jetzt noch kein flexibles Simulationssystem, welches Simulationen mit einer sehr hohen Genauigkeit ermöglicht und gleichzeitig Eigenschaften von PVNN wie zum Beispiel geringe Aktivität, spärliche Vernetzung und Verzögerungen zwischen Neuronen zur Optimierung der Simulation selber nutzt. Zwar lassen sich mit einigen Simulationssystemen wie z. B. GENESIS [BowBee1998] detaillierte Simulationen mit

hoher Zeitauflösung durchführen, jedoch sind diese Simulationen aufgrund des verwendeten Zeitscheibenverfahrens[1] auf kleinere Netzwerke beschränkt. Für große Netze pulsverarbeitender Neuronen erweist sich eine Simulation nach dem Zeitscheibenverfahren entweder als ungünstig oder sogar als praktisch gar nicht durchführbar, da alleine durch Vergröbern der Zeitauflösung, d.h. eine Vergrößerung der Zeitscheiben, akzeptable Simulationszeiten erreicht werden können.

Gesucht wird daher ein Simulationssystem, welches mit einer feinen Zeitauflösung, also hoher Genauigkeit, eine akzeptable Simulationsdauer für große PVNN aufweist. Außerdem sollte das Simulationssystem nur wenige Einschränkungen im Hinblick auf die Einbindung verschiedenster Neuronenmodelle aufweisen, da zwar einige Modelle für pulsverarbeitende Neuronen existieren, es jedoch Gegenstand vieler Untersuchungen ist, wie deren Parameter zu wählen sind. Auch neue Berechnungsmodelle, die aufgrund neuer Erkenntnisse aus der Neurophysiologie entwickelt werden, sollten eingebunden werden können. Ebenso sollte das Simulationssystem beliebige Vernetzungen der Neuronen zu Netzwerken gestatten.

In dieser Arbeit werden die Eigenschaften von PVNN untersucht und eine daraufhin optimierte Lösung in Form des Simulationssystems SPIKELAB vorgestellt. SPIKELAB implementiert eine ereignisgetriebene und verteilte Simulation, welche die Eigenschaften pulsverarbeitender neuronaler Netze, wie z. B. die Verzögerung zwischen zwei Neuronen, explizit behandelt und ein dadurch gegebenes Optimierungspotential ausschöpft. Trotz der in SPIKELAB eingesetzten Optimierungen können nahezu beliebige Berechnungsmodelle für Neuronen implementiert werden, da nur sehr wenig einschränkende Rahmenbedingungen durch das Simulationssystem selbst vorgegeben werden.

Da für den Transfer der Ereignisse in der verteilten, ereignisgetriebenen Simulation verschiedene Protokolle und Bibliotheken in Frage kommen, wird in dieser Arbeit untersucht, welches System sich am besten für die Verteilung auf einem heterogenen Netzwerk von Rechnern eignet.

Schließlich wurde in dieser Arbeit untersucht, in welcher Weise eine solche Simulation durch Hardware beschleunigt und wie analoge und digitale Hardware in die Simulation eingebunden werden kann. Die Ergebnisse dieser Untersuchungen sind in die Entwicklung der Beschleunigungshardware RACER eingeflossen, welche Bestandteil des SPIKELAB Simulationssystems ist.

1.2. Kapitelübersicht

Die Kapitel „Kapitel 3 - Simulation pulsverarbeitender neuronaler Netze“, „Kapitel 4 - Das SPIKELAB“ und „Kapitel 6 - RACER - Hardware zur Beschleunigung der Simulation und Integration von pulserzeugender Hardware“ stellen die Kernabschnitte dieser Arbeit dar, wobei die ersten beiden die Softwarerealisation des Simulationssystems erörtern und letzteres dessen Hardwarebeschleunigung behandelt. Die Software- und Hardwarekapitel fußen auf dem Kapitel „Kapitel 2 - Pulsverarbeitende neuronale Netze“ und lassen sich weitgehend unabhängig voneinander lesen. Abschnitt 2.1.6 bietet eine kompakte Zusammenfassung des Grundlagenkapitels, mit deren Hilfe die

[1] Beim Zeitscheibenverfahren werden alle Knoten des Netzes in jedem Zeitschritt der Simulation aktualisiert.

Ausführungen der folgenden Kapitel nachvollzogen werden können. Bei der alleinigen Lektüre des Hardwarekapitels könnte Abschnitt 3.3 hilfreich sein.

Kapitel 2 - Pulsverarbeitende neuronale Netze

Dieses Kapitel erläutert Grundlagen und insbesondere die zugrunde liegenden neurophysiologischen Annahmen, die zu den wesentlichen Designentscheidungen im vorgestellten Simulationssystem geführt haben.

Der Leser, der mit den neurophysiologischen Hintergründen vertraut ist, kann dieses Kapitel überspringen und bei Bedarf auf die komprimierte Zusammenfassung in Abschnitt 2.1.6 zurückgreifen.

Kapitel 3 - Simulation pulsverarbeitender neuronaler Netze

Das Kapitel bietet das theoretische Rüstzeug, auf dem das folgende Kapitel über die Realisation des Simulationssystems aufbaut.

Im Anschluss an die Vorstellung grundlegender Simulationsverfahren in Abschnitt 3.1, stellt Abschnitt 3.2 die Voraussetzungen für eine sequentielle, ereignisgetriebene Simulation pulsverarbeitender neuronaler Netze dar. Es werden die Modellbildung beleuchtet und die Anforderungen an die einzelnen Teilbereiche dieser Simulationsmethode beschrieben. Ein mit der ereignisgetriebenen bzw. verteilten Simulation vertrauter Leser findet in dem Abschnitt 3.3 eine Zusammenfassung der verwendeten Optimierungen. In Abschnitt 3.4 werden verschiedene Nachrichtensysteme für die Verteilung einer Simulation vorgestellt und im Hinblick auf ihre Eignung für die ereignisgetriebene und verteilte Simulation miteinander verglichen und bewertet. Abschließend werden in Abschnitt 3.5 verwandte Arbeiten vorgestellt.

Kapitel 4 - Das SPIKELAB

Dieses Kapitel stellt das Simulationssystem SPIKELAB vor, welches im Rahmen dieser Arbeit entwickelt wurde. Nach einem Überblick über das Gesamtsystem und einer kurzen Erläuterung der einzelnen Komponenten, wie der automatischen Partitionierung, der graphischen Oberfläche mit 3D-Visualisierung und der Anbindung an die beschleunigende Hardware, folgt in Abschnitt 4.2 eine Beschreibung der Softwarearchitektur des Simulators.

In Abschnitt 4.3 wird anhand von verschiedenen Beispielnetzen quantitativ die Leistungsfähigkeit des Simulationssystems SPIKELAB sowohl für den rein sequentiellen als auch für den verteilten Betrieb des Simulators untersucht und mit einem bereits verfügbaren Simulationssystemen verglichen.

Kapitel 5 - Hardware für die Simulation pulsverarbeitender neuronaler Netze

Abschnitt 5.1 zeigt Möglichkeiten zur Hardware-basierten Beschleunigung der ereignisgetriebenen und verteilten Simulation auf. Abschnitt 5.2 gibt einen Überblick über aktuelle Konzepte und Realisationen von Simulationshardware für PVNN.

Kapitel 6 - RACER - Hardware zur Beschleunigung der Simulation und Integration von pulserzeugender Hardware

Abschnitt 6.1 stellt die Architektur und Realisierung der RACER-Hardware mit ihrer Softwareumgebung vor. RACER dient in SPIKELAB sowohl der Beschleunigung der Simulation, als auch der Integration externer, pulserzeugender Hardwarekomponenten. Die Leistungsfähigkeit der Hardware wird in Abschnitt 6.2 durch einen quantitativen Vergleich mit der Softwarerealisation dargestellt.

Kapitel 7 - Zusammenfassung und Ausblick

Abschließend werden die zuvor aufgeführten Ergebnisse der einzelnen Kapitel zusammengefasst, bewertet, und ein Ausblick auf weitere Entwicklungsmöglichkeiten und Einsatzgebiete des Simulationssystems gegeben.

Kapitel 2 - Pulsverarbeitende neuronale Netze

Modelle pulsverarbeitender neuronaler Netze lassen sich nach einem Exkurs in die Welt der Hirnforschung leichter verstehen. Die Darstellung neurophysiologischer Erkenntnisse in dieser Arbeit beschränkt sich auf Eckdaten, die der Leser benötigt, um die nachfolgenden Kapitel nachvollziehen zu können.

Die Betrachtung des Gehirns als komplexes Netzwerk von pulsverarbeitenden Neuronen kann Hinweise auf funktionale Einheiten und deren Vernetzungsstrukturen liefern. Sie steckt den Rahmen für die Simulation von Netzwerken mit vergleichbarer Größe und Eigenschaft.

Technisch wichtige Einzelheiten, wie zum Beispiel die Parameter der Neuronenmodelle, begründen sich ebenfalls aus neurophysiologischen Erkenntnissen und dienen nicht zuletzt auch als Argumentationsbasis für die Auslegung einzelner Aspekte der Simulation. Dieser neurophysiologische Exkurs wird mit Hinblick auf eine technische Umsetzung durchgeführt. Er schließt mit einer tabellarischen Übersicht wichtiger Parameter und einer Zusammenfassung der wichtigsten Informationen über die Struktur der Netzwerke.

Nach der Vorstellung von neurophysiologischen Grundlagen, wendet sich die Arbeit der Modellbildung für pulsverarbeitende Neuronen und Netzwerke zu. Die vorgestellten Modelle unterscheiden sich im Wesentlichen durch den Grad der Abstraktion, welcher mit der Modellierung erreicht wird. Das Spektrum reicht dabei von den komplexen und detailreichen biologienahen Modellen bis hin zu den einfachen Ratenmodellen. Für technische Anwendungen werden in der Regel einfache Neuronenmodelle eingesetzt, da mehrere Neuronen zu einem Netzwerk zusammengeschlossen werden und diese in einer vertretbaren Rechenzeit simuliert werden müssen.

Da die Interpretation der zugrunde liegenden Informationscodierung untrennbar mit den Modellbildungen der Neuronen und Netzwerke verbunden ist, geht es um die Frage, ob pulsverarbeitende neuronale Netze einen Ratencode verwenden, ob allein der Zeitpunkt des einzelnen Pulses entscheidend ist oder ob eine Mischung von beiden Codierungsarten vorliegt. Natürlich spielt die Informationscodierung auch eine entscheidende Rolle bei der Entwicklung von geeigneten Lernalgorithmen (→ [Zell1994]). Für eine Ratencodierung existieren zum Beispiel bereits einige Lernalgorithmen, für einen Pulscode hingegen ist dies noch Gegenstand der Forschung. Deshalb werden die Ansätze vorgestellt, mit denen man der Dynamik Herr werden möchte, die es dem biologischen Vorbild ermöglicht, sich unterschiedlichsten Situationen anzupassen und neue Lösungen aus bestehendem Wissen zu entwickeln.

2.1. Das Gehirn als Modell für neuronale Netzwerke

2.1.1. Die Struktur des Gehirns

Nach heute weit verbreiteten Annahmen lässt sich das Gehirn als Zusammenspiel mehrerer Strukturen verstehen, die sowohl im Hinblick auf ihre Funktion als auch ihres Aufbaus unterschiedlich sind.

Die größte Struktur im menschlichen Gehirn, welche sich zuletzt entwickelt hat, ist die Großhirnrinde (Cerebraler Cortex). Abbildung 1 ist zu entnehmen, dass sich diese Struktur beinahe über alle darunter liegenden Strukturen erstreckt, welche nicht nur die Verbindung zum Rückenmark bilden, sondern auch wesentliche Funktionen des menschlichen Organismus, wie motorische Abläufe und die Produktion von Hormonen steuern. Einen größeren Raum nimmt hierunter das Kleinhirn (Cerebellum - Cerebellarer Cortex) ein, „welches vermutlich als erste Struktur für die sensomotorische Koordination verantwortlich war“ ([Thompson1994] S. 28).

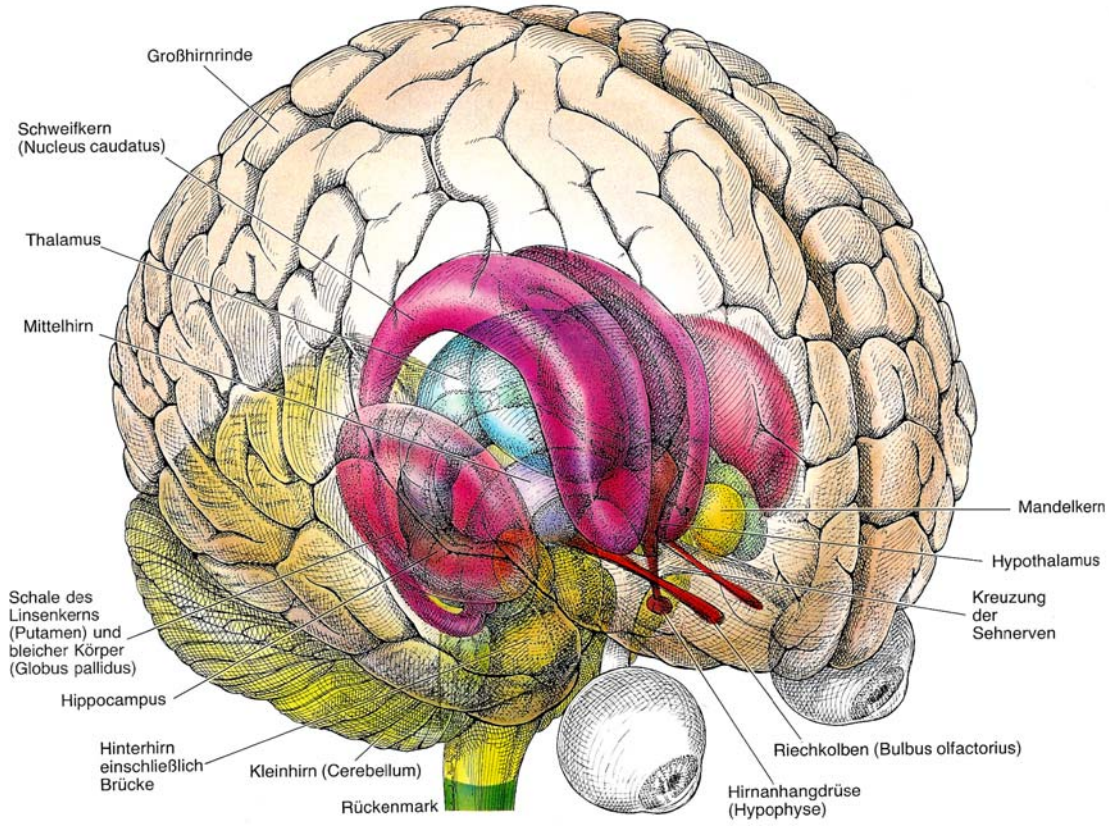


Abbildung 1: Aufbau des Gehirns (aus [Spektrum1988])

So unterschiedlich die verschiedenen Strukturen des Gehirns auch sind, so gleichartig ist jedoch ihr prinzipieller Aufbau, denn in jeder der Regionen finden sich Nervenzellen (Neuronen), welche über verschiedene Arten miteinander verknüpft sind. Die Unterschiede ergeben sich aus den in der Struktur vorliegenden Neuronenarten als auch aus der Art und dem Grade der Vernetzung zwischen diesen Neuronen. Der cerebrale Cortex enthält zum Beispiel ca. 1.5×10^{10} Neuronen und ca. 10^{14} synaptische Verbindungen und der cerebellare Cortex ca. 5×10^{10} Neuronen und ca. 10^{13} synaptische Verbin-

dungen[2]. Im gesamten Gehirn befinden sich ca. 7×10^{10} bis 8×10^{10} Neuronen. Diese Zahlen zeigen, dass cerebraler und cerebellarer Cortex die meisten „Verarbeitungseinheiten“ auf sich vereinen. Im Vergleich dazu befinden sich im Thalamus, der ersten „Relaisstation“ des optischen Nervs, ca. 10^6 Neuronen.

Aufgrund der hohen Verknüpfungsdichte im cerebralen Cortex werden Informationen in dieser Struktur sehr wahrscheinlich hauptsächlich assoziativ verarbeitet, wohingegen beim cerebellaren Cortex davon ausgegangen werden kann, dass die lokale Verarbeitung von Informationen das vorherrschende Prinzip darstellt.

Dem Cerebellum wird vornehmlich die Aufgabe zugeschrieben, die Motorik zu koordinieren und eine wichtige Rolle beim Lernen einzunehmen. Bei der Großhirnrinde lassen sich Bereiche identifizieren, die auf bestimmte Aufgaben spezialisiert sind. Diese Bereiche weisen zwar keine scharfen Grenzen auf und kein Gehirn gleicht dem anderen, jedoch lässt sich zumindest der Schluss ziehen, dass unterschiedliche Informationen parallel in verschiedenen, örtlich beschränkten Bereichen der Großhirnrinde verarbeitet werden. Abbildung 2 gibt einen Überblick über Lage und Ausdehnung einiger dieser Bereiche.

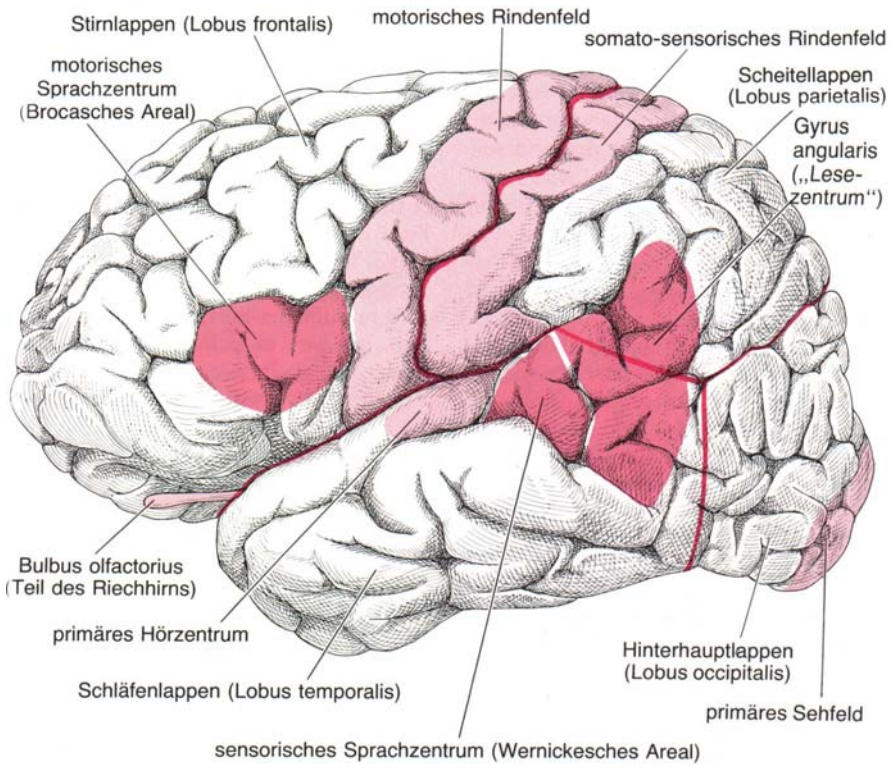


Abbildung 2: Funktionale Bereiche der Großhirnrinde (aus [Spektrum1988]).

Dem motorischen bzw. dem somato-sensorischen Rindenfeld können sogar bestimmte Körperteile zugeordnet werden. Die Aufgaben dieser Rindenfelder können als verzerrte Abbildung des gesamten Körpers, als so genannter *Homunkulus* dargestellt werden (Abbildung 3). Außerdem haben neurophysiologische Untersuchungen zu Tage gefördert, dass jede Verbindung im Gehirn eine systembedingte Verzögerung aufweist. Sie ist abhängig von der Dicke des die Neuronen verbindenden Axons, dessen gegebenenfalls vorhandenen Ummantelung mit Myelin und der Verzögerung, die an der

[2] [Schüz1995], [Haug1986], [BraSch1991] und [Braitenberg1977].

synaptischen Endigung des Axons vorliegt. Solche Zeitverzögerungen können eine effiziente Zusammenarbeit entfernter Neuronen verhindern und sie sind ein weiterer Hinweis auf eine Spezialisierung benachbarter Neuronen. Wollte man nun durch dickere Axone eine schnellere Fortleitung erreichen, so müssten aufgrund der Volumenvergrößerung des Gehirns die Axone auch wieder länger werden, wodurch die Verzögerung wiederum größer werden würde usw. Für das menschliche Gehirn müsste zum Beispiel das Volumen 50% größer sein, um eine Verdoppelung der Signalgeschwindigkeit zwischen den beiden Hemisphären zu erreichen ([RinDotDem1994] und [Schüz1995]).

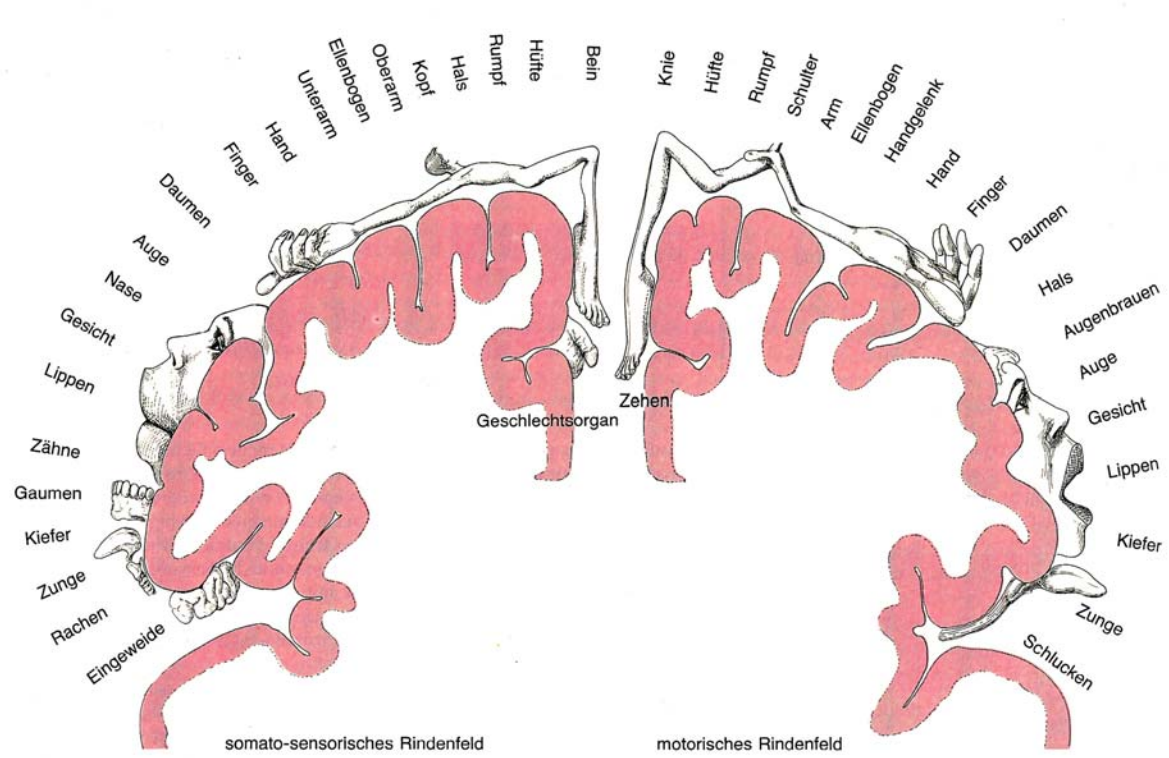


Abbildung 3: Homunkulus - Karte der Körperteile auf dem somato-sensorischen und dem motorischen und Rindenfeld (aus[Spektrum1988])

Das Gehirn besteht jedoch nicht nur aus den Neuronen und ihren Verbindungen, sondern zu etwa 50% aus Gliazellen, deren Funktion nicht endgültig bestimmt ist. Sie sind einerseits das strukturelle Gerüst für das Gehirn, andererseits erzeugen diese Zellen die Myelinhülle[3], welche die Axone von Neuronen umschließt. Sie nehmen überschüssige Neurotransmitter auf, die im Bereich der Synapsen entstehen, und füllen Leerräume auf, die durch abgestorbene Neuronen entstehen. Eine Informationsfortleitung durch Aktionspotentiale, wie sie bei Neuronen existiert, gibt es nach bisherigen Erkenntnissen bei Gliazellen nicht[4]. Die Gliazellen können also gewissermaßen als Versorgungsnetzwerk verstanden werden. Das Gehirn enthält ca. 4×10^{10} bis 5×10^{10} Gliazellen [Haug1986].

[3] Eine Myelinhülle ist eine Ummantelung der Nervenfaser aus einem Fett-Eiweiß-Gemisch, welche eine schnellere Signalfortleitung ermöglicht (siehe auch Kapitel 2.1.4, erster Absatz (Myelinisierung) .

[4] Laut einer Pressemeldung der Universität Leipzig vom 05.03.1999 wurde nachgewiesen, dass die Gliazellen an den Lernvorgängen im Gehirn teilhaben.

2.1.2. Vernetzung

Wenn man einen Schnitt durch den cerebralen Cortex betrachtet (Abbildung 4), kann man sechs verschiedene Schichten erkennen, in denen die Neuronen angeordnet sind. Neben diesen Schichten erkennt man in der Mitte der Abbildung Säulenanordnungen, und zwar sowohl bei der Anordnung der Neuronen als auch bei dem Verlauf der verbindenden Fasern, der im rechten Teil der Abbildung dargestellt ist. Diese beiden Aspekte dominieren die Struktur des cerebralen Cortex, wobei die Neuronen innerhalb einer Schicht keiner besonderen Ordnung folgen (Abbildung 5a). Im linken Teil derselben Abbildung sind die Erscheinungsformen typischer Neuronen der jeweiligen Schicht zu sehen.

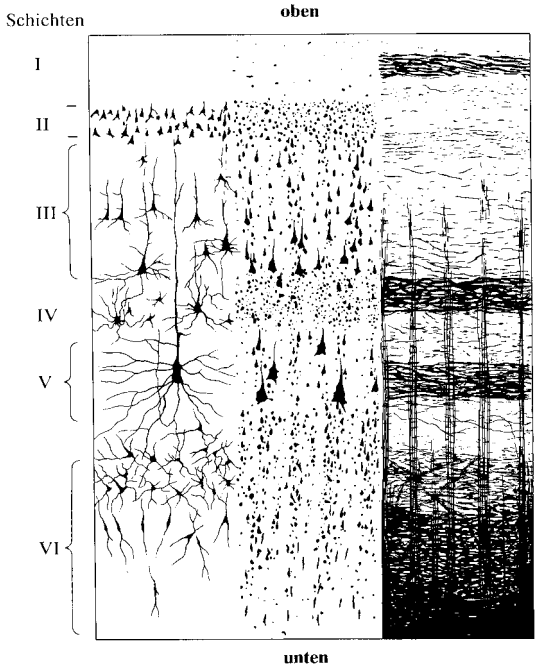


Abbildung 4: Schichten des cerebralen Cortex (aus [Thompson1994])

Der cerebellare Cortex hingegen besitzt drei, wesentlich stärker strukturierte Schichten. Insbesondere die Vernetzung ist im Cerebellum wesentlich strukturierter als im cerebralen Cortex. In der obersten Schicht verlaufen lange Axone parallel zur Oberfläche und dehnen sich dort weit aus. Senkrecht dazu sind jeweils in einer Fläche weit verästelte Dendriten angeordnet.

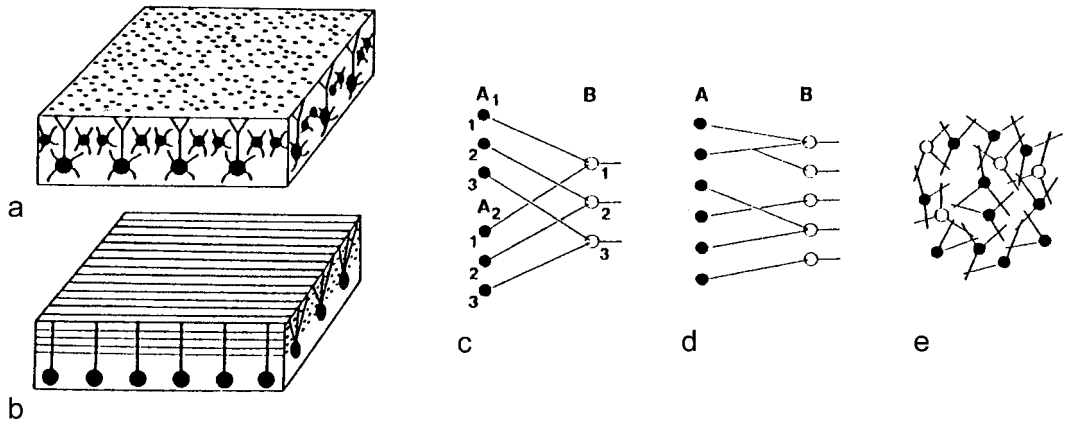


Abbildung 5: Neuronenanordnung und Vernetzungsarten (aus [Schüz1995])

Diese Struktur ist in Abbildung 5b schematisch dargestellt. Darüber hinaus sind im Cerebellum Neuronen eines Typs jeweils mit einem bestimmten anderen Typ verbunden.

Nach [Schüz1995] können im Gehirn drei prinzipielle Vernetzungsarten unterschieden werden. Die am stärksten strukturierte Variante definiert spezifische Verknüpfungen nicht nur zwischen bestimmten Neuronentypen, sondern auch zwischen individuellen Neuronen. Eine solche Situation liegt zum Beispiel im visuellen System der Fliege vor [Schüz1995, Braitenberg1977] (Abbildung 5c). Die zweite, weniger restriktive Variante definiert spezifische Verknüpfungen zwischen bestimmten Neuronentypen, wobei das individuelle Neuron keine Rolle spielt. Diese Situation findet sich im Cerebellum, da hier eine Vernetzung zwischen bestimmten Neuronentypen vorliegt (Abbildung 5d). Im cerebralen Cortex hingegen überwiegt, neben der durch Schicht- und Säulenanordnung vorgegebenen, eine zufällige Vernetzung unter den Neuronen (Abbildung 5e).

Würden die Verknüpfungen zwischen allen Neuronen in Form einer Matrix aufgetragen, so würde diese Matrix spärlich besetzt sein, da selbst im stark vernetzten cerebralen Cortex ein Neuron mit lediglich ca. $1{,}4*10^{-4}$ ‰ aller Neuronen verbunden ist. Ein Neuron ist nicht mit allen anderen, sondern mit einem Bruchteil aller Neuronen verknüpft. Dennoch besitzt ein Neuron im cerebralen Cortex durchschnittlich 7.000 Verbindungen zu benachbarten Neuronen.

2.1.3. Das Neuron und Neuronenarten

Wie alle tierischen Zellen hat jedes Neuron eine Zellmembran, welche den Zellinhalt und den Zellkern umschließt. Aus dem Zellkörper eines Neurons, dem ***Soma***, ragen als Fortsätze ein ***Axon*** und meist mehrere ***Dendriten*** heraus. Das Axon dient dazu, ein Neuron mit seinen Nachfolgern zu verbinden. Das Axon kann sich verzweigen und bildet Endigungen, so genannte Synapsen[5], entweder an einem Dendrit, oder direkt am Soma eines nachfolgenden Neurons. Manche Axone enden auch wieder auf dem Axon anderer Neuronen.

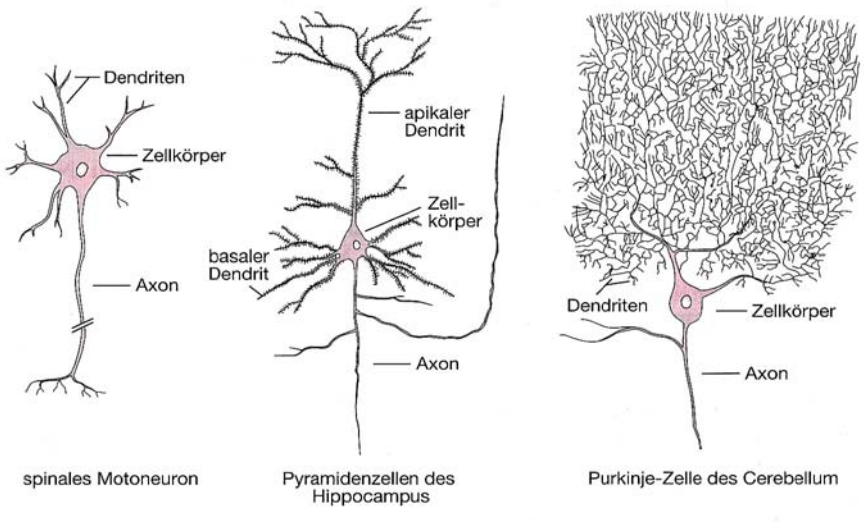


Abbildung 6: Neuronenarten (nach Ramón y Cajal, 1933 - aus [KanSchJes1995])

[5] Charles Sherrington führte 1897 [FosShe1897] den Begriff Synapse für die axonale Endung ein.

Die Dendriten sammeln gewissermaßen die Eingaben vieler anderer Neuronen. Die Formenvielfalt der Neuronen ist hauptsächlich durch die sehr unterschiedliche Ausprägung der Dendriten bestimmt. In Abbildung 6 werden drei verschiedene Neuronenarten gezeigt. Das mittlere Neuron, die Pyramidalzelle, ist die mit ca. 70% im cerebralen Cortex am häufigsten auftretende Neuronenart. Diese Neuronenart wirkt im Allgemeinen exzitatorisch, also erregend auf ihre Nachfolger, und besitzt sowohl kurze als auch lange Verbindungen[6]. Knapp weitere 25% werden durch die hier nicht abgebildeten Sternzellen gestellt, welche kurze, sternförmige Dendriten aufweisen und eher inhibitorisch, also hemmend auf ihre Nachfolger wirken.

Die Purkinje-Zelle tritt im cerebellaren Cortex auf und ermöglicht durch ihre extreme Verästelung Eingabe über ca. 150000 bis 200000 Synapsen.

Trotz dieser sehr verschiedenen Erscheinungsformen folgen alle Zellen dem gleichen Funktionsprinzip. Vereinfacht lässt sich das Funktionsprinzip einer Nervenzelle folgendermaßen beschreiben: Der Dendritenbaum sammelt die über die Synapsen einlaufenden Signale und leitet diese an das Soma weiter. Im Soma werden diese Signale in einem einfachen, nichtlinearen „Rechenschritt“ verarbeitet. Das Axon leitet schließlich das Ausgangssignal, welches in der Regel aus einem oder mehreren Spannungspulsen besteht, an nachfolgende Neuronen weiter.

Sofern keine Signale von anderen Neuronen vorliegen, befindet sich das Soma im Ruhezustand, der durch das Ruhepotential charakterisiert wird. Dieses Ruhepotential wird durch Ionenpumpen in der Zellmembran aufrechterhalten. Die Pumpen befördern nämlich Natrium-Ionen (Na^+-Ionen) aus der Zelle heraus und durch die Semipermeabilität der Zellmembran gelangen Kalium-Ionen (K^+-Ionen) in die Zelle hinein. Dadurch entsteht im Ruhezustand ein Überschuss an Na^+-Ionen im Außenraum der Zelle und ein Überschuss an K^+-Ionen im Inneren der Zelle. Die daraus resultierenden Konzentrationsgradienten verursachen ein elektrisches Membranpotential, welches bei Muskelfasern und Nervenzellen zwischen –100 mV und –55 mV liegt. Das Zellinnere ist also negativ gegenüber dem Zelläußeren geladen.

Eingehende Signale, welche exzitatorisch auf das Neuron wirken, depolarisieren die Zellmembran, so dass das Membranpotential positiver wird. Sofern nun die depolarisierende Wirkung von außen einen kritischen Bereich von –60 mV bis –45 mV überschreitet, öffnen sich spannungsabhängige Natriumkanäle in der Zellmembran, wodurch das Potential rasch bis zu einem Maximalwert von 30 mV ansteigt. Dieser Anstieg dauert bei Nerven- und Muskelzellen lediglich 0.2-0.5 ms [Schmidt1987]. Etwas langsamer reagieren die ebenso spannungsabhängigen Kaliumkanäle und öffnen sich, wenn sich die Natriumkanäle schließen. Damit endet der Puls und das Potential fällt wieder ab. Die längere Reaktionszeit der Kaliumkanäle führt dazu, dass das Potential über das Ruheniveau hinaus negativer wird. Man spricht in diesem Zusammenhang von einem Nachpotential oder einer Hyperpolarisierung der Zelle, welche eine erneute Aktivierung des Neurons erschwert. Diese Phase wird als relative *Refraktärphase* bezeichnet und schließt sich der absoluten Refraktärphase an, die ca. 2 ms andauert und unmittelbar nach dem Maximum des Aktionspotentials einsetzt. In der absoluten Refraktärphase kann das Neuron auch bei anhaltender Erregung die kritische Schwelle nicht mehr überschreiten. Reicht eine Erregung hingegen nicht aus, um die kritische Schwelle zu überschreiten, so bleibt das Neuron inaktiv. Daher wird das Prinzip, nach dem die Entste-

[6] Im Cortex der Maus sind die Axone z. B. durchschnittlich 10-40 mm lang und die Dendriten 4 mm. [Schüz1995]

hung eines Aktionspotentials abläuft, auch *Alles-oder-Nichts-Prinzip* genannt [Adrian1926], [Adrian1932]. Die beschriebenen Abläufe sind in Wirklichkeit erheblich komplizierter (siehe auch [LliSug1980], [Llinas1988] und [TsiLipMad1988]), müssen an dieser Stelle jedoch nicht weiter vertieft werden.

Neben dem typischen Verlauf eines Aktionspotentials, wie in Abbildung 7, gibt es noch viele verschiedene Varianten. Das Nachpotential kann z. B. bis zu 100 ms andauern [ConGutPri1982],[LanAda1986] oder auch 10-50 ms depolarisierend wirken [WonPri1981], das heißt eine erneute Erregung des Neurons wird in diesem Fall begünstigt.

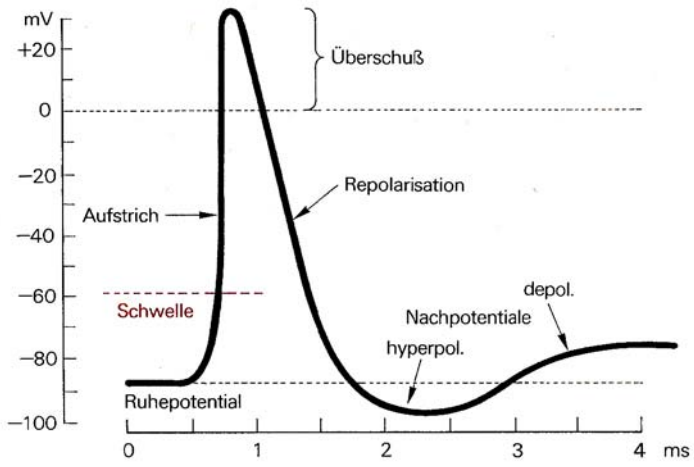


Abbildung 7: Aktionspotential (aus [Schmidt1987])

In Abbildung 8 sind drei verschiedene Verläufe dargestellt. Hier ist zu erkennen, dass das Nachpotential beim Muskelnerv der Ratte depolarisierend ist und dass vor allem die Dauer des Aktionspotentials sehr unterschiedlich ausfallen kann. Der Puls einer Nervenzelle der Katze dauert lediglich 1 ms, wohingegen der Puls am Herzmuskel etwa 300 ms andauert.

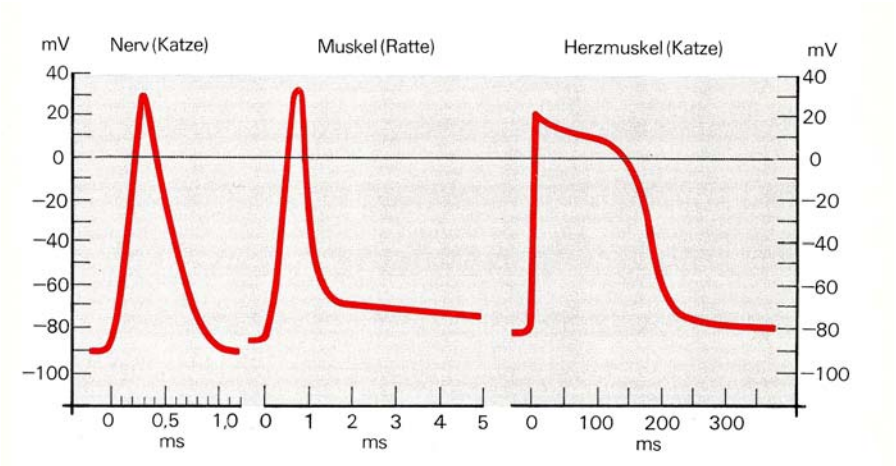


Abbildung 8: Formen des Aktionspotentials (aus [Schmidt1987])

Obwohl Aktionspotentiale eine recht unterschiedliche Ausprägung haben können, codiert ihre Form keine Informationen, die wesentlich für die Kommunikation zwischen den Zellen wären. Sie ist vielmehr systemspezifischer Natur und stellt verschiedene Arten der Ansteuerung dar, die sich aus dem spezifischen Milieu ergeben, in dem die Zelle

operiert. Da die Aktionspotentiale nach dem *Alles-oder-Nichts-Prinzip* ausgelöst werden, ist die entscheidende Information darin enthalten, ob ein Aktionspotential vorliegt, und vor allem, wann es vorliegt, jedoch nicht darin, welche Form es aufweist. Eine Untersuchung aus [RieWarRuy1997] stützt diese Annahme. Im Rahmen dieser Untersuchung konnte an einer Nervenzelle aus dem Fliegengehirn nachgewiesen werden, dass die Form des Aktionspotentials über mehrere Messungen hinweg reproduzierbar ist (Abbildung 9).

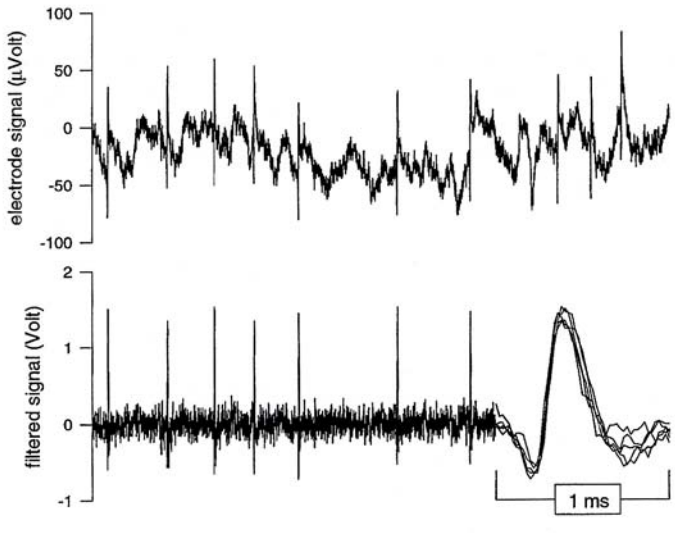


Abbildung 9: Messung des Aktionspotentials einer Nervenzelle aus einem Fliegengehirn: Oben der abgeleitete Spannungsverlauf; Unten: Der Bandpassgefilterte Verlauf, wobei auf der rechten Seite fünf Aktionspotentiale überlagert auf einer gestreckten Zeitachse dargestellt sind. (aus [RieWarRuy1997])

Wie die Aktionspotentiale fortgeleitet und schließlich über axonale Endigungen an die Synapsen übertragen werden, wird im nächsten Abschnitt genauer betrachtet.

2.1.4. Die Synapse – Signalfortleitung

Die Mechanismen, die der Fortleitung des Aktionspotentials zugrunde liegen, entsprechen denen, die für ihr Zustandekommen verantwortlich sind, das heißt die Fortleitung ist ein aktiver und kein passiver Prozess. Die schlagartige Depolarisation der Membran pflanzt sich über das Axon fort, bis sie die axonale Endigung erreicht. Die Form des Aktionspotentials verändert sich auf diesem Weg kaum oder gar nicht, daher weisen die Aktionspotentiale an einer Synapse immer die gleiche Gestalt auf. Die Geschwindigkeit, mit der sich das Aktionspotential ausbreitet, hängt davon ab, wie dick das Axon ist und ob es myelinisiert ist oder nicht. Je dicker das Axon ist, desto schneller geht die Fortleitung vonstatten, da der Leitungswiderstand mit wachsendem Querschnitt sinkt. Die Ummantelung einer Nervenfaser aus dem Fett-Eiweiß-Gemisch (Myelinisierung), setzt die Kapazität der Membran herab- und ihren Widerstand herauf. Dadurch verringern sich die Ableitungs- und Umladungsverluste. Bei Wirbeltieren sind alle Nervenfasern myelinisiert, die eine höhere Leitungsgeschwindigkeit als 3m/s aufweisen [Schmidt1987]. So ummantelte Fasern können Leitungsgeschwindigkeiten bis über 100 m/s erreichen. An Nervenfasern wurden Leitungsgeschwindigkeiten zwischen weniger als 1 m/s und mehr als 100 m/s gemessen. Die Länge eines Axons kann von wenigen Millimetern bis weit über einem Meter betragen [Schmidt1987].

Bei den axonalen Endigungen werden seltenere elektrische von den häufigeren chemischen Synapsen unterschieden. Bei der elektrischen Synapse ist das Axon fest mit dem

Nachfolgeneuron entweder am Dendrit oder direkt am Soma verbunden. Der Abstand zwischen präsynaptischer und postsynaptischer Membran beträgt ca. 3,5 nm und die Übertragung erfolgt nahezu verzögerungsfrei, da die Depolarisation der postsynaptischen Zelle durch Strominjektion erfolgt. Bei der chemischen Synapse ist die präsynaptische Membran durch einen 20-40 nm breiten Spalt von der postsynaptischen Membran getrennt. Das Aktionspotential löst in der Synapse des präsynaptischen Neurons chemische Überträgersubstanzen, so genannte Neurotransmitter aus. Dabei öffnen sich Vesikel und schütten den in ihnen enthaltenen Neurotransmitter aus. Die Freisetzung des Transmitters erfolgt also in diskreten Mengen. Der freigesetzte Neurotransmitter diffundiert durch den Spalt und bindet an spezifische Rezeptoren der postsynaptischen Membran. Ob die Zelle dadurch depolarisiert, also erregt, oder hyperpolarisiert, also gehemmt wird, hängt von der Art des Rezeptors ab.

Aus den verschiedenen Schritten, die bei der Übertragung an einer chemischen Synapse ablaufen, ergibt sich eine synaptische Verzögerung von 0.3 ms bis zu einigen Millisekunden[KanSchJes1995]. Trotz der langsameren Übertragung an einer chemischen Synapse hat diese gegenüber der elektrischen Synapse einen Vorteil im Hinblick auf ihre Effizienz. Sie kann nämlich verstärkend wirken, da ein relativ geringer Strom ausreicht, eine beträchtliche Menge Neurotransmitter auszuschütten und dadurch auch größere postsynaptische Zellen zu depolarisieren. Bei einer elektrischen Synapse sind hingegen relativ große Ströme notwendig, um die postsynaptische Zelle zu erregen. Zudem kann sich bei chemischen Synapsen die Effizienz stärker verändern als bei elektrischen Synapsen. Diese Eigenschaft spielt eine wichtige Rolle beim Erlernen von Verhaltensweisen.

Es gibt darüber hinaus Hinweise darauf, dass die Verzögerungen nicht nur systemspezifisch sind, sondern auch eine Bedeutung bei der Informationsverarbeitung haben [HunGluPal1998], [SteWil1997] und [Schmitt1999]. Ebenso existieren Theorien zu entsprechenden Lernverfahren, wie zum Beispiel in [GerKemHem1996] beschrieben.

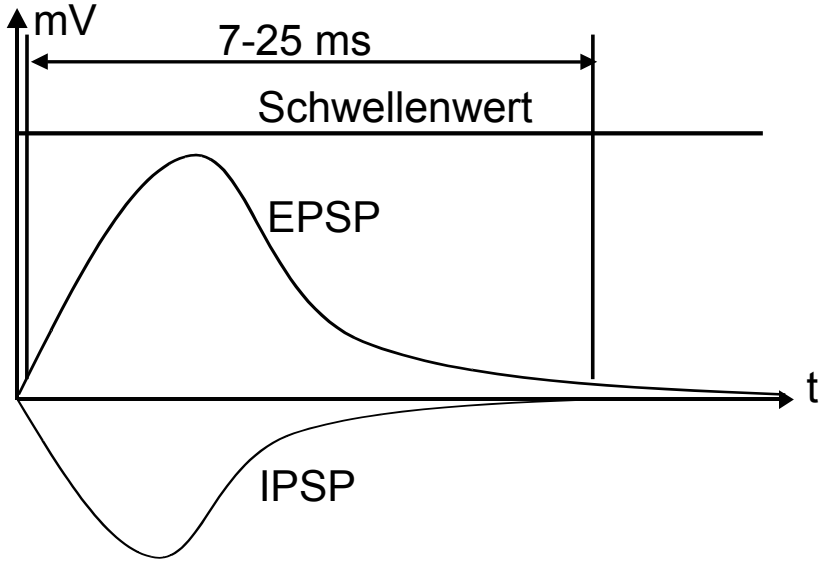


Abbildung 10: Verlauf des exzitatorischen und inhibitorischen Potentials

Die aus der synaptischen Übertragung resultierenden exzitatorischen (EPSP)[7] oder inhibitorischen (IPSP)[8] postsynaptischen Potentiale zeigen den in Abbildung 10 charakte-

[7] **EPSP**: **E**xzitatorisches **P**ost **S**ynaptisches **P**otential

ristischen Verlauf. Je nach Typ der Synapse erstreckt sich der Anstieg des postsynaptischen Potentials über 2-5 ms und das Abklingen benötigt 5-20 ms [BroJoh1983].

2.1.5. Aktivität

Aus dem Verlauf des Aktionspotentials eines Neurons ist ersichtlich, dass ein Neuron nicht beliebig kurz hintereinander ein Aktionspotential aussenden kann. Insbesondere die absolute Refraktärphase, die sich der Auslösung eines Aktionspotentials anschließt, verhindert einen Anstieg der Aktivität über eine maximale Rate. Da die Auslösung eines Aktionspotentials auch als das „Feuern“ des Neurons bezeichnet wird, nennt man diese Rate auch Feuerrate. Ausgehend von einer ca. 2 ms dauernden Refraktärphase würde eine maximale Feuerrate von 500 Pulsen pro Sekunde möglich sein. In neurophysiologischen Messungen wurden Aktivitäten zwischen wenigen Pulsen bis hin zu 600 Pulsen pro Sekunde gemessen, wobei nicht adaptive Neuronen eine höhere Rate (300 – 600 Hz) aufweisen als adaptive Neuronen (100 – 200 Hz) [McCConLig1985][JahLli1984]. Die Messergebnisse stehen im Einklang mit den theoretisch ermittelten Grenzen.

Durch die Beschränkung der Feuerrate eines einzelnen Neurons ist auch die durchschnittliche Aktivität eines Netzwerks von Neuronen beschränkt. Diese Aktivität ist jedoch noch weit niedriger als die durch die maximalen Feuerraten vorgegebene Schranke. Eine stark erhöhte Aktivität in Teilen des Cortex ist vielmehr ein Krankheitsbild und damit eine Fehlfunktion. So tritt z. B. eine solche Überaktivität im Zuge eines epileptischen Anfalls auf. Für eine optimale Auslegung der Simulation solcher Netzwerke wäre es nun wünschenswert, die durchschnittliche Gesamtaktivität im Gehirn zu kennen. Eine Messung der Gesamtaktivität ist jedoch mit heutigen Mitteln nicht durchführbar. Deshalb ist nur eine grobe Abschätzung möglich, die sich auf Messungen in eingegrenzten Bereichen des Cortex stützt (wie z. B. ca. 4 Pulse pro Sekunde und Neuron aus [KrüAip1988]). Auf ähnliche Größenordungen kann man aufgrund der sehr Umfangreichen Untersuchungen in [Abeles1982] und [Abeles1991] schließen.

Da jedoch bei lokal beschränkten Messungen in der Regel in den Bereichen gemessen wird, in denen auch eine Aktivität in Abhängigkeit des durchgeführten Experiments erwartet wird, kann ausgehend von solchen Messungen insgesamt von einer noch niedrigeren durchschnittlichen Aktivität im Cortex ausgegangen werden. Schätzungen gehen von weit weniger als einem Puls pro Sekunde und Neuron aus, d.h. durchschnittlich wird ein Neuron maximal einmal in der Sekunde aktiv.

2.1.6. Zusammenfassung der Eckdaten und tabellarische Übersicht

Aus den neurophysiologischen Modellvorstellungen lassen sich die folgenden, für die Simulation relevanten Eigenschaften zusammenfassen:

Das Gehirn stellt ein spärlich vernetztes Netzwerk einer enormen Anzahl von Verarbeitungseinheiten, den Neuronen dar. Die Neuronen tauschen dabei Informationen in Form von Pulsen, den Aktionspotentialen aus. Hierbei stellt die Existenz und der Zeitpunkt des Aktionspotentials die ausgetauschte Information dar, nicht jedoch die Form desselben (Alles-Oder-Nichts Prinzip). Insgesamt liegt eine sehr geringe durchschnittliche Aktivität in dem betrachteten System vor. Systembedingt findet die Fortleitung über das Axon und die Synapsen immer verzögert statt.

---

[8] **IPSP**: **I**nhibitorisches **P**ost **S**ynaptisches **P**otential

2.1.6.1. Gehirn

Anzahl der Neuronen	~7-8x10^{10}
Gesamtaktivität	unbekannt (Annahme: $\varnothing \leq 1$ Puls pro Sekunde und Neuron)

2.1.6.2. Cerebraler Cortex

Neuronen	~1,5x10^{10}
Verknüpfungen	~10^{14}
Neuronenarten	70% Pyramidalzellen und 25% Sternzellen
Vernetzung	In 6 Schichten mit Säulenstrukturen. Innerhalb der Schichten weitgehend unstrukturierte Verknüpfungen

2.1.6.3. Cerebellarer Cortex

Neuronen	~5x10^{10}
Verknüpfungen	10^{13}
Neuronenarten	Purkinjezellen, Kernerzellen, Golgizellen
Vernetzung	In 3 Schichten stark strukturierte vertikal/horizontal Verknüpfungen

2.1.6.4. Aktionspotential

	Typisch	**Maximal**
Dauer	1–2 ms	200–300 ms
Anstiegszeit	0,2–0,5 ms	
Abfallzeit	1–1,5 ms	200–290 ms
Maximalfrequenz	100-200 Hz bzw. 300-600Hz	1000 Hz
Pegel	Spitze: 30 mV / Ruhepot.: -60 bis –45 mV	30 mV

2.1.6.5. Postsynaptische Potentiale

	EPSP	**IPSP**
Dauer	15 ms	10–15 ms
Anstiegszeit	2–5 ms	2 ms
Abfallzeit	5–10 ms	5–10 ms
Ruhepotential	-15 mV	-80 mV

2.1.6.6. Synapsen

	elektrisch	**chemisch**
Abstand zum Dendrit/Soma	3–4 nm	20–40 nm
Adaptationsfähigkeit	gering	hoch
Verzögerung	~0	0.3 ms bis wenige ms
Leistungsbedarf	hoch	gering
Vorkommen	selten	oft

2.1.6.7. Axone

Längen	Wenige mm bis > 1m
Leitungsgeschw.	1 m/s – 100 m/s
Verzögerung	0,1 ms/cm – 10 ms/cm, je nach Leitungsgeschwindigkeit Die Spanne $1 \leq \Delta_{ax} \leq 5$ ms kann als typisch angesehen werden.
Bemerkung	Die Geschwindigkeit hängt vom Querschnitt und der Myelinisierung ab.

2.2. Modelle pulsverarbeitender Neuronen

Die Ausführungen in diesem Abschnitt stützen sich im Wesentlichen auf das erste Kapitel in [MaaBis1998]. Die Modelle pulsverarbeitender Neuronen lassen sich übersichtlich nach verschiedenen Abstraktionsebenen darstellen:

Auf der mikroskopischen Ebene werden unterschiedliche Ionenkanäle, die Durchlässigkeit der Zellmembran und verschiedenste chemische Botenstoffe modelliert. Diese sehr biologienahen Modelle haben ihren Ursprung in den Formeln, die von Hodgkin und Huxley aufgestellt wurden. Abschnitts-Modelle (Englisch: *compartment model*) bilden zusätzlich die räumliche Struktur des betrachteten Neurons ab. Die Modellierung durch ein Abschnitts-Modell kommt einer räumlichen Zerlegung des Neurons gleich, wobei jeder Abschnitt durch einen äquivalenten elektronischen Schaltkreis modelliert wird.

Auf dem nächst höheren Abstraktionsniveau werden Neuronen als homogene Funktionseinheiten aufgefasst, die Pulse generieren, sofern eine ausreichende Erregung vorliegt. Hier findet keine detaillierte Modellierung der elektrochemischen Vorgänge statt und die räumliche Struktur wird ebenso wenig abgebildet. Diese Modelle werden gemeinhin als Integrate-and-Fire-Modelle bezeichnet. Es gibt jedoch sehr viele unterschiedliche Implementierungen, die diese Bezeichnung für sich beanspruchen. Die im Folgenden vorgestellte Form des Integrate-and-Fire-Neurons stellt gewissermaßen die einfachste Grundform eines solchen Neurons vor. Alle gängigen Modifikationen des Integrate-and-Fire-Neurons lassen sich mit Hilfe des Spike-Response-Modells (SRM) [DomHemSch1995] modellieren. Das Eckhorn-Modell ([EckBauJor1988], [EckReiArn1989]) fällt im Grunde auch in die Klasse der Integrate-and-Fire-Modelle, besitzt jedoch entgegen allen anderen Modellen noch einen oder mehrere so genannte "Linking" Eingänge, welche multiplikativ auf die Summe der postsynaptischen Potenti-

ale wirken. Das Modell nimmt damit eine Sonderstellung unter den Integrate-und-Fire-Modellen ein.

Ratenmodelle stehen auf einer noch höheren Abstraktionsebene als die Spike-Response-Modelle, da nicht mehr der einzelne Puls modelliert wird, sondern die Rate der Pulse, die ein Neuron über einen bestimmten Zeitraum generiert. Mit dieser Art der Modellierung kann insbesondere die Ansteuerung der Muskeln im Körper nachgebildet werden, da die Stärke, mit der ein Muskel kontrahiert, im Wesentlichen von der applizierten Pulsrate der steuernden Neuronen abhängt. Effekte, die durch synchrones bzw. asynchrones Feuern der einzelnen Neuronen entstehen, lassen sich jedoch mit solch einer Codierung nicht mehr untersuchen. Die zeitlichen Aspekte der Informationsverarbeitung können daher nur unzureichend oder gar nicht modelliert werden. Deshalb werden diese Modelle im Folgenden nicht weiter betrachtet.

2.2.1. Hodgkin-Huxley-Modell

Hodgkins und Huxleys Untersuchungen am Riesenaxon des Tintenfischs stellen gewissermaßen den Ursprung aller Modelle pulsverarbeitender Neuronen dar, die auf Basis von Leitwerten beschrieben werden. Sie leiteten aus sehr umfangreichen Untersuchungen vier Differentialgleichungen (Gleichung 1,3,4 und 5) ab, die es ihnen gestatteten, bei entsprechender Wahl der Parameter die experimentellen Daten mit den Modellgleichungen in Einklang zu bringen.

Das kapazitive Verhalten der Zellmembran beschrieben sie durch

$$C\frac{du}{dt} = -\sum_k I_k + I(t), \qquad \text{Gl. (1)}$$

wobei I(t) ein von außen eingeprägter Strom und $\sum_k I_k$ die Summe aller Ionenströme durch die Zellmembran ist. Im Hodgkin-Huxley-Modell werden drei verschiedene Ionenkanäle bzw. Arten von Strömen unterschieden. Es werden Natriumkanäle für den Strom der Natriumionen (Index Na), Kaliumkanäle für den Strom der Kaliumionen (Index K) und unspezifische Leckstromkanäle (Index L) unterschieden[9]. Somit setzt sich der Summenstrom wie folgt zusammen:

$$\sum_k I_k = g_{Na} m^3 h (u - V_{Na}) + g_K n^4 (u - V_K) + g_L (u - V_L) \qquad \text{Gl. (2)}$$

Die Parameter g_i sind die Leitwerte der entsprechenden Kanäle und die Potentiale V_i sind Konstanten, die bestimmen, welches Vorzeichen der Summenstrom trägt, in Abhängigkeit der aktuellen Spannung u, die über der Membran abfällt. Mit Hilfe der zusätzlichen Variablen n, m und h wird die Spannungsabhängigkeit der Ionenkanäle beschrieben:

$$\dot{m} = \alpha_m(u)(1-m) - \beta_m(u)m \qquad \text{Gl. (3)}$$

$$\dot{n} = \alpha_n(u)(1-n) - \beta_n(u)n \qquad \text{Gl. (4)}$$

[9] Englisch: Natrium → Sodium und Kalium → Potassium

$$\dot{h} = \alpha_h(u)(1-h) - \beta_h(u)h \qquad \text{Gl. (5)}$$

Die Faktoren $\alpha_i(u)$ und $\beta_i(u)$ sind empirisch bestimmte Funktionen der Spannung *u*, die aus den Messungen am Riesenaxon des Tintenfischs ermittelt wurden. Entsprechend den angegebenen Gleichungen kann für das Hodgkin-Huxley-Modell ein elektrisches Ersatzschaltbild angegeben werden (Abbildung 11).

Die Spannungsabhängigkeit und das zeitliche Verhalten dieses Modells lassen sich leichter verstehen, wenn die Gleichungen 3 bis 5 nach [GerHem1994] bzw. [MaaBis1998] in die Form

$$\dot{x} = -\frac{1}{\tau(u)}\left[x - x_0(u)\right] \qquad \text{Gl. (6)}$$

gebracht werden. Dabei steht *x* für *m*, *n* bzw. *h* und $x_0(u)$ ist ein asymptotischer Wert, dem sich die Variable *x* mit der Zeitkonstanten $\tau(u)$ annähert, sofern die Spannung *u* auf einen konstanten Wert eingestellt wird. Somit ergibt sich für den asymptotischen Wert der Zusammenhang $x_0(u) = \frac{\alpha_x(u)}{\left[\alpha_x(u) + \beta_x(u)\right]}$ und für die Zeitkonstante $\tau(u) = \left[\alpha_x(u) + \beta_x(u)\right]^{-1}$. Sigmoidale Verläufe von *m*, *n* und *h*, sowie die unterschiedlichen Zeitkonstanten führen dazu, dass durch eine Erregung von außen, welche die Spannung über der Zellmembran anhebt, das Membranpotential weiter ansteigt. Dieser Anstieg wird durch den Faktor *h* bei höheren Spannungswerten gestoppt. Der Faktor *n* steigt zur gleichen Zeit, bedingt durch eine größere Zeitkonstante, verzögert an. Dies führt dazu, dass das Membranpotential durch eine äußere Erregung rasch auf einen hohen Wert ansteigt und sich anschließend mit einem negativen Überschwingen langsam wieder dem Ruhepotential nähert. Das Modell von Hodgkin und Huxley beschreibt also eine Folge von Aktionspotentialen, welche entstehen, wenn der in die Zelle konstant eingeprägte Strom größer als ein kritischer Wert I_θ ist. Wird der Strom *I(t)* über den Wert I_θ hinaus erhöht, so erhöht sich die Rate, mit der die Aktionspotentiale erzeugt werden.

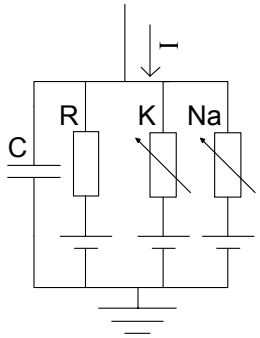


Abbildung 11: Ersatzschaltbild des Hodgkin-Huxley-Modells

Das Hodgkin-Huxley-Modell wurde ursprünglich für die Beschreibung der Form und des zeitlichen Verhaltens bei der Fortleitung eines Aktionspotentials auf dem Tintenfischaxon entworfen. Um die Entstehung des Aktionspotentials am Soma beschreiben zu können, wurden zu den Gleichung 1 bis 5 vergleichbare Gleichungen verwendet

(vgl. [MaaBis1998]). Im Wesentlichen wurden zusätzliche Ionenkanäle in die Beschreibung integriert, deren Eigenschaften ebenfalls aus experimentellen Daten bestimmt wurden. Dies betraf insbesondere sich langsam ändernde Ionenströme, welche im Zusammenhang mit den Adaptationseigenschaften der Neuronen stehen. Eine so erweiterte Beschreibung stellt den Grundbaustein der Abschnitts-Modelle dar.

2.2.2. Abschnitts-Modelle – compartment models

Neben der Erweiterung der Gleichungen von Hodgkin und Huxley um langsam veränderliche Ionenströme, die das adaptive Verhalten der Neuronen beschreiben, führen die Abschnitts-Modelle vor allem eine Modellierung der räumlichen Struktur ein. Das Neuron wird dabei nicht als punktförmige Verarbeitungseinheit betrachtet, sondern das Neuron und seine Fortsätze werden abschnittsweise durch äquivalente elektrische Schaltkreise beschrieben (Abbildung 12). Hierbei werden insbesondere die Dendriten mit einem wesentlich größeren Detailgrad modelliert.

Grundsätzlich können die einzelnen Abschnitte mit einem beliebigen Detailgrad modelliert werden. Die Berechnung der Modelle im Rahmen einer Simulation ist in der Praxis jedoch durch die verfügbare Rechenzeit beschränkt. Daraus folgt, dass bei detaillierter Modellierung der Neuronen die Größe der simulierbaren Netzwerke beschränkt ist. Daher werden Abschnitts-Modelle auch vorwiegend für detaillierte Simulationen einzelner Neuronen oder kleinerer Neuronengruppen verwendet.

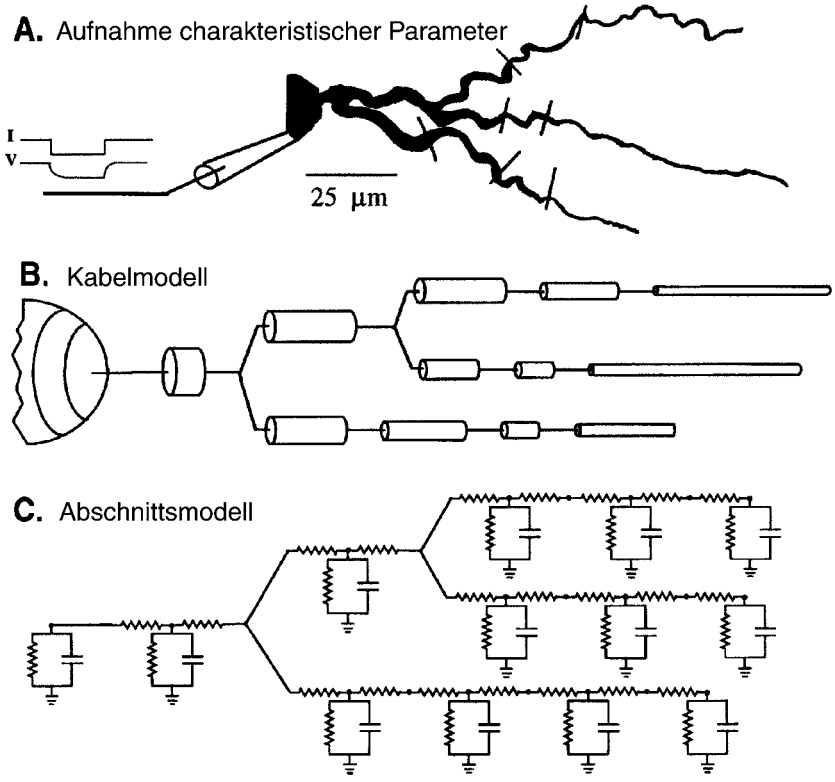


Abbildung 12: Prinzipielles Vorgehen bei der Erstellung eines Abschnitts-Modells. Aus neurophysiologischen Messungen werden Parameter für ein Kabelmodell gewonnen (A) und das Neuron wird mit seinen Fortsätzen durch kurze Kabelabschnitte modelliert (B). Diese Kabelabschnitte werden anschließend durch äquivalente elektronische Schaltkreise ersetzt (C). (aus [BowBee1998])

2.2.3. Integrate-and-Fire-Modell

Wie schon erläutert, handelt es sich beim Integrate-and-Fire-Modell um eine Spezialform des Spike-Response-Modells. An dieser Stelle wird die gängigste Darstellung eines Integrate-and-Fire-Modells beschrieben. Der grundlegende Schaltkreis des Integrate-and-Fire-Modells besteht aus der Parallelschaltung eines Kondensators mit einem Widerstand. Er wird durch einen Strom I(t) gespeist (Abbildung 13).

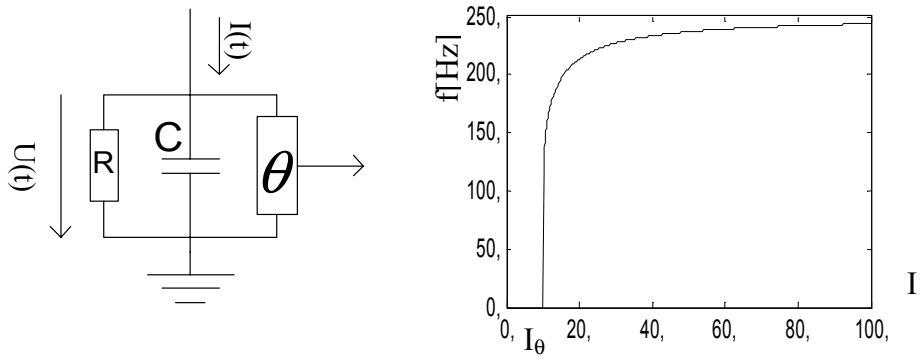


Abbildung 13: Schaltbild und Feuerrate eines Integrate-and-Fire-Neuron

Dieser Schaltkreis implementiert einen verlustbehafteten Integrator, welcher der Gleichung 7 folgt:

$$I(t) = \frac{u(t)}{R} + C\frac{du}{dt} \qquad \text{Gl. (7)}$$

u(t) ist hierbei die Spannung, die über dem Kondensator abfällt. Mit $\tau = RC$ als Zeitkonstante des Integrators lässt sich Gleichung 7 auch in der Form $\tau\frac{du}{dt} = -u(t) + RI(t)$ schreiben. In dieser Gleichung kann *u(t)* als das Membranpotential und τ als Membranzeitkonstante der zu modellierenden Zelle interpretiert werden. Erreicht das Membranpotential den Schwellenwert θ, so wird ein Ausgangspuls erzeugt und das Membranpotential auf einen Wert u_r zurückgesetzt. Der Zeitpunkt t_f, zu dem dies geschieht, wird auch Feuerzeitpunkt genannt, wobei $\lim_{\delta\to 0} u(t_f + \delta) = u_r$ gilt. Wird ein Ausgangspuls erzeugt, so schließt sich eine absolute Refraktärphase an, in der nicht weiter über den Eingangsstrom integriert wird. Ist die Dauer der Refraktärphase mit t_r gegeben, lässt sich die Feuerfrequenz des Neurons in Abhängigkeit des speisenden Stroms ausdrücken:

$$f(I) = \left[t_r + \tau \log\frac{I}{I - I_\theta} \right]^{-1} \qquad \text{Gl. (8)}$$

Durch eine geeignete Wahl der Parameter *R, C* und t_r lässt sich mit dem Integrate-and-Fire-Neuron das Feuerverhalten des Hodgkin-Huxley-Modells approximieren. Sofern die Zeitkonstante für die Umladung der Membran im Hodgkin-Huxley-Modell als der die Dynamik bestimmende Parameter gewählt wird, lassen sich die Modelle ineinander überführen.

2.2.4. Spike-Response-Modell

Sowohl das Integrate-and-Fire-Modell als auch das Hodgkin-Huxley-Modell werden auf Basis eines speisenden Stroms modelliert. Beim Spike-Response-Modell (SRM) basiert die Modellierung hingegen auf den Zeitpunkten der eingehenden Aktionspotentiale. Diese werden hierbei zu Pulsen ohne Ausdehnung gleich einer Deltafunktion idealisiert. Diese Art der Modellierung trägt den neurophysiologischen Beobachtungen Rechnung, nach denen die Aktionspotentiale, welche an einer Synapse eintreffen, nur marginale Unterschiede in ihrer Form aufweisen[10], so dass angenommen werden kann, dass die übermittelte Information im Zeitpunkt, zu dem das Aktionspotential auftritt, und nicht in der Form des Aktionspotentials steckt. Da jedoch, sobald ein solcher Puls eintrifft mitunter komplizierte Abläufe auf der postsynaptischen Seite der synaptischen Verbindung angestoßen werden und diese Abläufe zudem ein zeitlich veränderliches Verhalten aufweisen können, wird im Spike-Response-Modell jedem eingehenden Puls eine Eingangsfunktion $\varepsilon(t)$ zugeordnet. In der einfachsten Form dieses Modells ist vorgesehen, die zeitlich zueinander verschobenen und mit einem individuellen Gewicht w_{ij} bewerteten Eingangsfunktionen zu summieren und den Verlauf dieser Summe auf die Überschreitung eines gegebenen Schwellenwertes zu untersuchen. Wird der Schwellenwert überschritten, so löst dies einen Ausgangspuls aus, welcher wiederum eine Ausgangsfunktion $\eta(t)$ verursacht, die erlaubt, die reduzierte Erregbarkeit eines Neurons nach dem Feuern zu modellieren. Dieses Verhalten entspricht einem dynamischen Schwellenwert, wie er bei manch anderen Modellen zu finden ist. Das innere Summenpotential wird bei diesem Modell nicht wie beim Integrate-and-Fire-Modell zum Zeitpunkt des Feuerns zurückgesetzt, sondern alleine die Addition der Ausgangsfunktion führt dazu, dass das Neuron eine kurze Zeit gar nicht feuert und danach nur feuert, wenn es stärker angeregt wird. Grundsätzlich ist die Form der Ein- und Ausgangsfunktionen nicht vorgegeben, jedoch wird das Spike

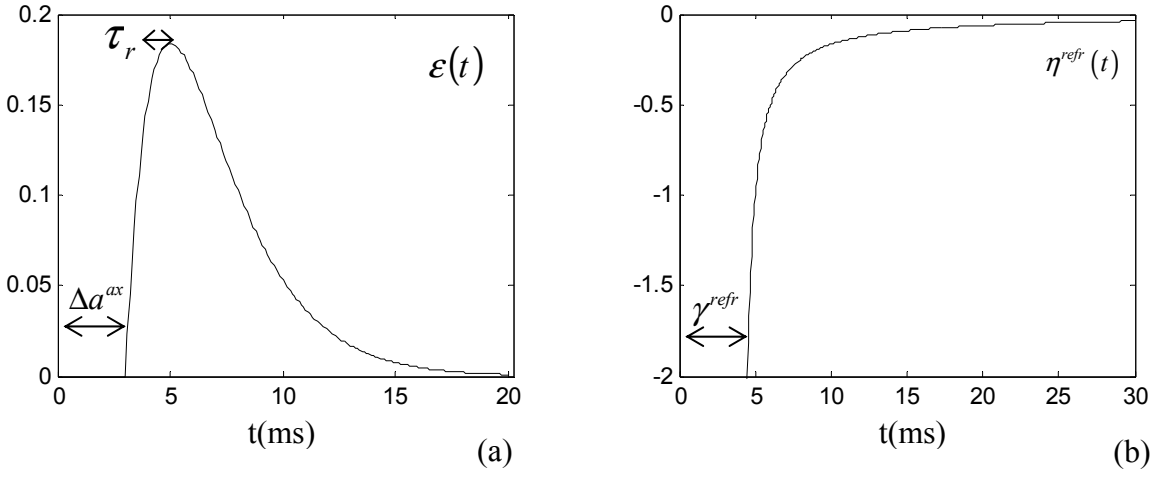


Abbildung 14: Typische Verläufe (a) der Eingangsfunktion (postsynaptisches Potential) und (b) der Ausgangsfunktion (Nachpotential) in einem Spike-Response-Modells.[11]

[10] Vgl. Messungen des Aktionspotentials: Abbildung 9

[11] Mit (a): Δa^{ax} = 3ms und τ_r = 2ms (mit $1 \le \Delta^{ax} \le 5ms$ und $2 \le \tau_r \le 5ms$) und (b): γ^{refr} = 4ms (mit $0 \le \gamma^{refr} \le 4ms$ und $\eta_0 \approx 1$ ms)

Response Modell in der Regel mit den in der Abbildung 14 gezeigten Funktionen modelliert. Die Eingangsfunktion fällt nach einem Anstieg rasch wieder auf Null ab. Zwar nähert sich die zur Modellierung verwendete Funktion ($\alpha(s) = \frac{1}{\tau_s} e^{\left(-\frac{s}{\tau_s}\right)}$ für s > 0), die so genannte Alphafunktion, dem Wert Null nur an, jedoch kann ihr Wert nach einer angemessenen Zeitdauer in einer Simulation zu Null gesetzt werden, wodurch sie das Summenpotential nicht mehr beeinflusst. Ebenso kann mit der Ausgangsfunktion verfahren werden, da diese sich von negativen Werten dem Wert Null annähert.

Der innere Zustand *h* des Modells wird durch die Summe der synaptischen Eingangsfunktion $h^{syn}(t)$ und die durch Aussendung von Pulsen entstandenen Ausgangsfunktion $h^{refr}(t)$ beschrieben:

$$h(t) = h^{syn}(t) + h^{refr}(t) \qquad \text{Gl. (9)}$$

wobei

$$h^{refr}(t) = \sum_f \eta^{refr}\left(t - t_i^f\right) \quad \text{Gl. (10)} \qquad \eta^{refr}(t) = \begin{cases} -\infty & \text{für } t \le \gamma^{refr} \\ \eta_0 / \left(t - \gamma^{refr}\right) & \text{für } t > \gamma^{refr} \end{cases} \quad \text{Gl. (11)}$$

und

$$h_i^{syn}(t) = \sum_j w_{ij} \sum_f \varepsilon\left(t - t_j^f\right) \qquad \text{Gl. (12)}$$

$$\varepsilon(t) = \begin{cases} 0 & \text{für } 0 \le t \le \Delta a^{ax} \\ \frac{\left(t - \Delta a^{ax}\right)}{\tau_r} \cdot e^{\frac{-\left(t-\Delta a^{ax}\right)}{\tau_r}} & \text{für } t > \Delta a^{ax} \end{cases} \qquad \text{Gl. (13)}$$

gilt. γ_{refr} ist die absolute Refraktärphase, in der das Neuron auch bei einer neuerlichen Erregung nicht feuert. Δa_{ax} steht für die axonale Verzögerung und τ_r steht für die Anstiegszeit der Eingangsfunktion. Für die Feuerzeitpunkte gilt der Zusammenhang $t_x^f \le t$, das heißt, der Feuerzeitpunkt muss vor dem aktuellen Betrachtungszeitpunkt liegen. Die Summe über *f* kann in der Praxis nach oben beschränkt werden, da wie eingangs erläutert, der Wert der Eingangs- bzw. Ausgangsfunktionen nach einer angemessenen Abklingdauer eine zu vernachlässigende Größe darstellt.

Im SRM ist noch eine Ergänzung um Rauscheinflüsse vorgesehen, die eine realistischere Modellierung des biologischen Vorbilds zulässt. Hierfür wird eine Feuerwahrscheinlichkeit $P_F(h, \delta t)$ während eines infinitesimal kleinen Zeitraums δt angegeben:

$$P_F(h; \delta t) = \tau^{-1}(h)\,\delta t \qquad \text{Gl. (14)} \qquad \text{mit} \qquad \tau(h) = \tau_0 e^{-\beta[h-\theta]} \qquad \text{Gl. (15)}$$

bzw. im diskreten Fall[12]: $$P_F(h; \Delta t) = 1 - e^{-\tau^{-1}(h)\Delta t} \qquad \text{Gl. (16)}$$

[12] Für die Herleitung sei auf [DomHemSch1994], S. 46 verwiesen.

Die Antwortzeit $\tau(h)$ ist bestimmt durch das innere Potential *h* und dessen Nähe zum Schwellenwert θ, wobei bei einem biologienahen Modell $\tau(h)$ lang sein sollte, solange $h < \theta$ gilt (um ein Feuern zu verhindern) und verschwindend gering für $h \geq \theta$.

Eine weitere Anpassung des SRM zur besseren Abbildung natürlicher neuronaler Netze bzw. die Modellierung bestimmter technischer Aspekte oder theoretischer Überlegungen kann durch die Verwendung anderer Eingangs- und Ausgangsfunktionen erfolgen. Dadurch lassen sich zum Beispiel nahezu beliebig komplexe Modelle für die Verarbeitung im Dendritenbaum des Neurons konstruieren. Je nach Komplexität der Modellierung muss jedoch deutlich mehr Rechenzeit in die Simulation des Neuronenmodells investiert werden. Dies ist insbesondere dann der Fall, wenn Funktionen verwendet werden, die analytisch nicht beschreibbar sind. Solche Funktionen werden durch Tabellen realisiert, die einerseits den Speicherbedarf für die Simulation deutlich erhöhen und andererseits in der Regel viele Fallunterscheidungen bei der Auswertung des Modells erfordern, so dass entsprechend mehr Befehle für die Berechnung des Modells ausgeführt werden müssen. Darüber hinaus kann das Modell natürlich noch durch weitere Funktionalitäten, wie z. B. ein Zurücksetzen des inneren Potentials nach dem Feuern erweitert werden.

In dieser Flexibilität liegt die Stärke des SRM, denn letztlich kann der Detailgrad der Modellierung in weiten Bereichen variieren. Fixiert ist lediglich der Zusammenhang, dass sich aufgrund eingehender Pulse eine Reaktion einstellt (Spike → Response), welche wiederum auch auf den Modellzustand zurückwirken kann. Wie die Auswirkungen des „Response“ aussehen, ist jedoch frei bestimmbar. Das Standardmodell modelliert mit Hilfe dieser „Grundregel“ recht gut ein natürliches Neuron. Bei einem Vergleich zwischen dem Hodgkin-Huxley-Modell und einer Nachbildung dieses Modells durch ein SRM konnte nachgewiesen werden, dass das SRM bei entsprechender Parameterwahl eine ca. 90-prozentige Übereinstimmung der Aktionspotentiale mit den Aktionspotentialen des Hodgkin-Huxley-Modell erreicht [MaaBis1998].

2.2.5. Eckhorn-Modell

Das Eckhorn-Modell geht auf verschiedene Arbeiten [EckBauJor1988], [EckReiArn1989] zurück. Es unterscheidet sich von den Spike-Response-Modellen durch eine andere Verarbeitung der eingehenden Pulse. Es sieht zwei funktional verschiedene Typen von Eingängen vor: Feeding-Eingänge und Linking-Eingänge. Bei beiden Eingangstypen löst ein eingehender Puls eine mit W_i gewichtete Antwortfunktion aus. Die Antwortfunktionen werden für jeden Typ summiert Gleichung 17 und 18, wobei bei den Linking-Eingängen der konstante Summand 1 hinzugefügt wird.

$$h^L(t) = 1 + \sum_{\tau_i^L \in F^L, \tau_i^L \leq t} W_i^L e^{-\alpha\left(t-\tau_i^L\right)} \quad \text{Gl. (17)} \qquad h^F(t) = \sum_{\tau_i^F \in F^F, \tau_i^F \leq t} W_i^F e^{-\beta\left(t-\tau_i^F\right)} \quad \text{Gl. (18)}$$

$F = \{\tau_1, \tau_2, ...\}$ ist die Menge der Feuerzeitpunkte aller Vorgängerneuronen.

Die beiden Summanden werden anschließend multiplikativ verknüpft. Durch den konstanten Summanden im Linking-Zweig wird erreicht, dass sich das Eckhorn-Modell wie ein Spike-Response-Modell verhält, solange kein Signal an einem der Linking-Eingänge anliegt. Treffen nun Eingangspulse an den Linking-Eingängen ein, so wird das über die Feeding-Eingänge eingehende Signal entsprechend der Gewichtungen der

Linking-Eingänge verstärkt oder abgeschwächt. Die Autoren [EckReiArn1989] beschreiben dieses Verhalten als Amplitudenmodulation eines Signals (Feeding) durch ein anderes (Linking). Wird durch die Anregung über die Feeding- und Linking-Eingänge der Schwellenwert ϑ überschritten, so wird ein Ausgangspuls erzeugt, welcher zudem dazu führt, dass eine Ausgangsfunktion ausgelöst wird. Durch diese Ausgangsfunktion besitzt das Eckhorn-Modell genauso wie das SRM einen dynamischen Schwellenwert.

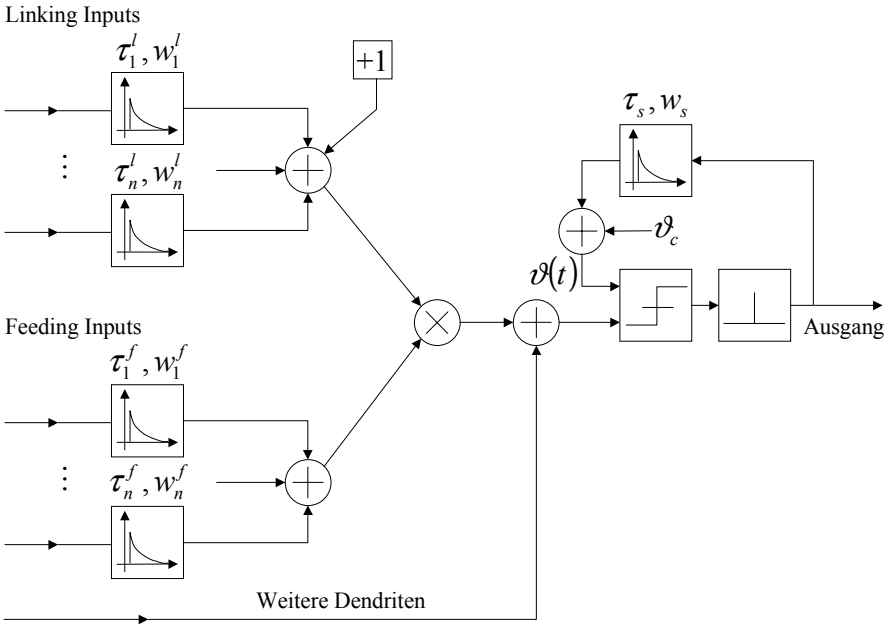


Abbildung 15: Aufbau des Eckhorn-Neuronenmodells

Für eine optimierte Auslegung der Simulation sind die Eigenschaften der Eingangs- bzw. Ausgangsfunktionen eines Modells von entscheidender Bedeutung. Beim Eckhorn-Modell weisen sowohl die Linking- und Feeding-Eingänge als auch die Ausgangsfunktion des Modells zum Zeitpunkt der Auslösung ihr Maximum auf und gehen dann exponentiell gegen Null (Abbildung 16). Der instantane Anstieg der Eingangsfunktion kann dazu führen, dass ein solches Neuron beim Eintreffen eines postsynaptischen Pulses sofort aktiv wird, also ein Aktionspotential aussendet, sofern durch den postsynaptischen Puls eine Eingangsfunktion mit positivem Vorzeichen ausgelöst wird und die Summation des Maximums der Eingangsfunktion mit dem aktuellen Membranpotential oberhalb des Schwellenwerts liegt. Eine Vorhersage, ob das Neuron aufgrund eines eingehenden Pulses feuert oder nicht, wie sie bei der ereignisgetriebenen Simulation notwendig ist, würde sich dadurch erheblich vereinfachen, da stets nur das Summenpotential zum Zeitpunkt des eintreffenden Pulses berechnet und mit dem Schwellenwert verglichen werden müsste.

Es kann jedoch der sehr seltene Fall eintreten, in dem zeitgleich postsynaptische Pulse mit unterschiedlichem Vorzeichen eintreffen, welche Eingangsfunktionen unterschiedlicher Zeitkonstante auslösen. Solche Signale würden sich anfänglich auslöschen. Im weiteren Verlauf kann es dann jedoch, bedingt durch die unterschiedlichen Zeitkonstanten, zu einem Anstieg des Potentials über den Schwellenwert kommen. Da dieser Fall potentiell auftreten kann, ist es nicht möglich, den instantanen Anstieg der postsynaptischen Eingangsfunktionen zur Optimierung in einer ereignisgetrieben Simulation heranzuziehen.

Der biologische Hintergrund des Modells ist durch Beobachtungen getragen, die ein funktional unterschiedliches Verhalten von Synapsen nahe dem Soma gegenüber denen auf den Dendritenenden motivieren. Demnach sollen dem Soma nahe Synapsen die Reaktion auf mehrere eingehende Signale an den Dendritenenden unterdrücken oder verstärken[13] können.

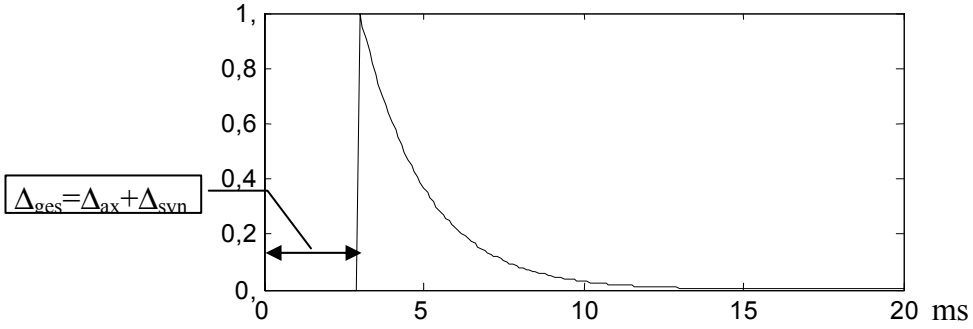


Abbildung 16: Die postsynaptische Eingangsfunktion des Eckhorn-Modells weist einen instantanen Anstieg auf und setzt nach der Summe Δ_{ges} aus axonaler und synaptischer Verzögerung ein.

Das Eckhorn-Modell wurde speziell für Netzwerke entwickelt, welche Merkmale aus der Netzeingabe extrahieren und diese durch Ausgangspulsfolgen repräsentieren. Die Merkmale werden durch bestimmte Frequenz- und Phasenbeziehungen der Ausgangspulsfolgen codiert. Besonderes Augenmerk wurde dabei auf eine Übereinstimmung mit neurophysiologischen Messungen natürlicher neuronaler Netze gelegt, so dass diese Messungen gut durch das künstliche Netz nachgestellt werden können.

Die Anregungsfunktion des Modells ist mit der Menge der Feuerzeitpunkte $F = \{\tau_1, \tau_2, ...\}$ der Vorgängerneuronen und der Menge $\hat{F} = \{\hat{\tau}_1, \hat{\tau}_2, ...\}$ der Zeitpunkte, zu denen das Neuron selbst feuert, durch

$$h(t) = \underbrace{\left(1 + \sum_{\tau_i^L \in F^L, \tau_i^L \le t} W_i^L e^{-\alpha\left(t-\tau_i^L\right)}\right) \sum_{\tau_i^F \in F^F, \tau_i^F \le t} W_i^F e^{-\beta\left(t-\tau_i^F\right)}}_{x(t)} - \sum_{\hat{\tau}_i^L \in \hat{F}^L, \hat{\tau}_i^L \le t} V e^{-\gamma(t-\hat{\tau}_i)} \qquad \text{Gl. (19)}$$

gegeben. Gleichung 19 gilt für einen Linking-Zweig und einen Feeding-Zweig. Als Erweiterung des Modells sind jedoch noch weitere Dendritenbäume, mit entsprechenden Linking- und Feeding-Zweigen, vorgesehen. Die Beiträge der einzelnen Dendritenbäume werden summiert. Daher folgt mit Gleichung 19 und der Menge D der Dendritenbäume für den allgemeinen Fall:

$$h(t) = \sum_D x(t) - \sum_{\hat{\tau}_i^L \in \hat{F}^L, \hat{\tau}_i^L \le t} V e^{-\gamma(t-\hat{\tau}_i)} \qquad \text{Gl. (20)}$$

2.2.6. Verlustlose Integrate-and-Fire-Modelle

Anstelle einer verlustbehafteten Integration wird in den verlustlosen Integrate-and-Fire-Modellen der Eingangswert lediglich summiert und, wenn die Summe einen Schwellenwert überschreitet, ein Puls ausgesendet.

[13] Synapsen nahe dem Soma oder direkt auf dem Soma wirken überwiegend inhibitorisch.

Dient eine Deltafunktion zur Informationsübermittlung zwischen diesen Neuronen, so löst diese beim Eintreffen einen Rechteckpuls endlicher Ausdehnung aus. Die Dauer und die Höhe des Rechteckpulses bestimmen den Beitrag, den dieser Eingangspuls zum Summenpotential liefert. Da die Integration nicht verlustbehaftet ist, kann zudem noch ein Sättigungsgrad für die Eingänge definiert werden. Dadurch kann der Beitrag, den ein einzelner Eingang zum Summenpotential beisteuert, auf einen Maximalwert beschränkt werden.

Wird ein Puls ausgelöst, so wird das Summenpotential zurückgesetzt, und es folgt eine absolute Refraktärphase, in der das Neuron auch durch erneute Eingangspulse nicht zum Feuern gebracht werden kann.

Da lediglich eine einfache Summation zur Bestimmung des Summenpotentials durchzuführen ist, stellt die Verwaltung der zeitlich zueinander verschobenen, eingangsseitigen Rechteckpulse den hauptsächlichen Aufwand in der Berechnung des Modells dar. Damit ist der Geschwindigkeitsvorteil eingeschränkt, der durch die Verwendung dieses Modells erreicht werden kann. Zudem werden durch diese Modellierung die neurophysiologischen Zusammenhänge nur vage skizziert. Deshalb kann das Modell vor allem dazu dienen, technische Implementierungen pulsverarbeitender neuronaler Netze zu simulieren, in denen eine solche Vereinfachung, mit Hinblick auf eine technische Realisierung, bewusst vorgenommen wird.

2.2.7. Stochastische Modelle

In biologischen neuronalen Netzen bewirkt ein eingehendes Aktionspotential nicht immer die Freisetzung von Vesikeln. Vielmehr muss von einer Wahrscheinlichkeit ausgegangen werden, mit der Vesikel durch ein Aktionspotential ausgelöst werden. Darüber hinaus stellt der Schwellenwert, ab dem ein Ausgangspulses erzeugt wird, keine scharfe Grenze dar. Dieser Aspekt wird zum Beispiel im Spike-Response-Modell durch eine Feuerwahrscheinlichkeit berücksichtigt, welche von der Nähe des Summenpotentials zum Schwellenwert abhängt. Bei stochastischen Modellen für pulsverarbeitende neuronale Netze wird das Neuron als Zufallsprozess mit bestimmten Eigenschaften bezüglich der Verteilung von Feuerzeitpunkten aufgefasst. Die Modellierungen finden in unterschiedlichen Abstraktionsebenen statt, beginnend mit der detaillierten Modellierung einzelner Vesikelauslösungen bis hin zu allgemeinen Statistiken über das Feuerverhalten.

Im Hinblick auf eine ereignisgetriebene Simulation ist es schwierig, die zeitliche Konsistenz zu gewährleisten, da das stochastische Modell zu beliebigen Zeitpunkten aktiv werden kann. Das Modell folgt keinem einfachen Reiz-Reaktionsschema, sondern kann auch von sich aus aktiv werden. Als einfachste Lösung bietet sich in diesem Fall an, die Zufälligkeit „einzuplanen“, indem eben diese Modelle mit einem gewissen zeitlichen Vorlauf „zufällige“ Ereignisse erzeugen, wodurch die zeitliche Konsistenz der Simulation gewährleistet werden kann. Schwierig wird eine solche Behandlung, wenn der Zufallsprozess zudem von den Eingaben seiner Vorgänger abhängt, die wiederum vom Zufallsprozess abhängen. Solche Ringbezüge erhöhen den Verwaltungsaufwand beträchtlich.

2.3. Modelle der Informationscodierung und -verarbeitung

2.3.1. Modelle der Codierung

In [MaaBis1998] werden Ratencodes und Spikecodes unterschieden. Die in dieser Arbeit betrachteten pulsverarbeitenden neuronalen Netzwerke verwenden einen Spikecode, welcher insbesondere für die Informationsverarbeitung im Cortex ein plausibleres Codierungsmodell als der Ratencode ist.

2.3.1.1. Ratencodes

Bei den Ratencodes wird über die Zahl der von den Neuronen ausgesendeten Pulse gemittelt. Es werden drei verschiedene Arten der Mittelung unterschieden: Die zeitliche Mittelung, die Mittelung über unterschiedliche Experimente und die räumliche Mittelung.

Bei der zeitlichen Mittelung werden die Pulse, die an dem Eingang eines Neurons auftreten bzw. von einem Neuron generiert werden, über ein festes Zeitfenster gemittelt, welches in Abhängigkeit der Neuronentypen und Stimuli gewählt wird. Typische Zeitfenstergrößen betragen 100ms oder 500ms. Diese Art der Codierung konnte erfolgreich für die Beschreibung einiger sensorischer und motorischer Experimente verwendet werden. Betrachtet man jedoch Erkennungsleistungen, die in weniger als 100ms ablaufen, oder in 30-40ms ablaufende Reaktionen, so stellt die zeitliche Mittelung für solche Situationen keine plausible Codierung dar.

Die Mittelung über unterschiedliche Experimente dient zur Bestimmung der Aktivität in einem Netzwerk, da die Dichte der Pulse bestimmt wird. Sie ist jedoch sehr wahrscheinlich kein Codierungsschema der neuronalen Interaktion, da die Wiederholung von Stimuli in ein und derselben Weise in der Natur praktisch nicht auftritt.

Bei der räumlichen Mittelung wird über die Aktivität so genannter Neuronenpopulationen gemittelt. Die Populationen sind Neuronenmengen, die ihrer Funktion nach gruppiert werden können. Der Vorteil der räumlichen Mittelung besteht darin, dass im Gegensatz zur zeitlichen Mittelung auch schnelle Veränderungen erfasst werden können.

2.3.1.2. Spikecodes

Spikecodes basieren auf den Zeitpunkten von Pulsen. Dabei kann die Codierung auf verschiedenen Grundlagen beruhen:

Wird allein der erste Puls nach einer abrupten Veränderung des Eingangsstimulus betrachtet, wird auch von der so genannten *Time-to-first-Spike* Codierung gesprochen. Eine solche Codierung ist plausibel, da einerseits schnelle Reaktionen (30-40ms) auf Reize beobachtet werden und andererseits die Verarbeitung mehrerer Pulse während einer solchen Reaktion schon aufgrund der Refraktärphase der Neuronen unwahrscheinlich ist.

Bei der Phasencodierung wird die Phasenlage einzelner Pulse bezüglich der Oszillation einer globaler Variablen, zum Beispiel der Aktivität ganzer Neuronenpopulationen, bestimmt.

Die synchrone Aktivität von einzelnen Neuronen oder ganzen Neuronengruppen kann ein Hinweis darauf sein, dass die betreffenden Neuronen das gleiche Problem bearbeiten – in dem gerade betrachteten Verarbeitungsschritt also zusammenwirken.

Darüber hinaus können auch bestimmte Pulsabfolgen, so genannte *Spikepattern*, Information tragen. Diese Pattern werden durch eine charakteristische Folge von Pulsen bestimmt, die in einer festen räumlichen und zeitlichen Relation zueinander stehen.

2.3.2. Lernen in pulsverarbeitenden neuronalen Netzen

Lernen in neuronalen Netzen ist die gezielte, problemabhängige Modifikation der Synapseneffizienz. Die Modifikationen werden dabei nicht von außen vorgenommen, sondern werden durch die Verarbeitung der angelegten Stimuli bedingt[14].

Die bekanntesten Mechanismen aus der Neurophysiologie sind die *short term potentation* (STP) und die *long term potentation* (LTP) bzw. *long term depression* (LTD). In beiden Fällen, STP und LTP, wird die Effizienz der Synapse verstärkt, wenn eine Folge präsynaptischer Pulse an einer Synapse eintrifft. Wobei jedoch bei der Langzeitpotenzierung gleichzeitig eine ausreichende Aktivierung der Zelle vorliegen muss, andernfalls tritt eine Langzeitdepression (LTD) ein, welche die Effizienz der Synapse reduziert. LTP und LTD haben eine Wirkdauer von Stunden bis zu Tagen, die STP hingegen im Minutenbereich. Experimentelle Befunde [MarLueFro1997] belegen zudem, dass die von der empfangenden Zelle ausgesendeten Aktionspotentiale auf die Synapsenmodifikation rückwirken. Dadurch hängt die Synapsenmodifikation von der zeitlichen Korrelation der prä- und postsynaptischen Aktionspotentiale ab.

Die Lernverfahren für künstliche neuronale Netze gehen auf das Hebbsche Postulat zurück [Hebb1949], welches besagt, dass eine synaptische Verbindung zwischen zwei gleichzeitig aktiven Neuronen verstärkt wird. Als Erweiterung des Hebbschen Postulats werden Lernverfahren für künstliche neuronale Netze zusätzlich mit einem Abklingterm versehen, durch den die synaptische Effizienz geschwächt wird, sofern keine gleichzeitige Aktivierung vorliegt.

Hauptsächlich unterscheiden sich die Lernverfahren durch den Einsatz unterschiedlicher Bewertungsfunktionen sowohl für die Abhängigkeit von der Aktivierung als auch für die Abhängigkeit von der zeitlichen Abfolge des eintreffenden und ausgehenden Pulses einer Zelle. Für die Simulation ist es darüber hinaus entscheidend, ob das Lernverfahren auf zur Synapse lokalen Informationen basiert oder zusätzlich Informationen benötigt, die entweder von benachbarten Neuronen bereitgestellt werden müssen oder global zum Netzwerk sind. Zur Synapse lokale Informationen sind die Zeitpunkte der präsynaptischen Pulse und alle postsynaptischen Informationen, wie z. B. die Aktivierung und die Zeitpunkte der ausgehenden Pulse des zugehörigen Neurons. Nicht lokal sind hingegen die Aktivierung der präsynaptischen Neuronen oder Informationen über die Aktivität von Neuronenpopulationen oder des gesamten Netzwerks. Eine detailliertere Diskussion der unterschiedlichen Verfahren ist in [Schäfer2000] zu finden.

[14] Für künstliche neuronale Netze gibt es auch eine Reihe von überwachten Lernverfahren, bei denen die Modifikationen auf globalen Informationen beruhen und von außen vorgenommen werden.

Kapitel 3 - Simulation pulsverarbeitender neuronaler Netze

Führt man das menschliche Gehirn als Beispiel für mögliche Netzwerkstrukturen ins Feld, so wird deutlich, dass von einem System ausgegangen werden kann, welches sehr viele Verarbeitungseinheiten besitzt und eine, auf die Gesamtzahl aller Verarbeitungseinheiten bezogene, geringe durchschnittliche Vernetzung aufweist. Hinzu kommt eine durchschnittlich sehr geringe Aktivität im Netzwerk, d.h. Neuronen feuern nur in sehr großen Abständen.

Für solche Netzwerke ist es ungeschickt, eine Simulation zu verwenden, bei der in äquidistanten Schritten der Zustand eines jeden Neurons bestimmt wird und damit in jedem Zeitschritt der Zustand des Gesamtsystems berechnet werden muss. Dies würde selbst für sehr grobe Zeitauflösungen enorme Rechenzeiten erfordern. Die Zeitauflösung kann jedoch nicht beliebig grob gewählt werden, da sich deren Wahl an den systemspezifischen Zeitkonstanten für den Anstieg des Aktionspotentials, der postsynaptischen Potentiale und an den axonalen und synaptischen Verzögerungen orientieren muss. Entsprechende Überlegungen in [BowBee1998] führen für die dort verwendete Zeitscheibensimulation zu einer Auflösung von 10 µs.

Der Rechenzeitbedarf einer solchen Simulation kann reduziert werden, wenn nur die Neuronen berechnet werden, die an der Informationsverarbeitung teilnehmen. Da die Zustände der Neuronen jedoch nur von einem Zeitschritt zum nächsten bestimmt werden, müssen alle Neuronen betrachtet werden, die in Zukunft ihren Schwellenwert überschreiten könnten. Im einfachsten Falle sind dies alle Neuronen, die entweder ein Summenpotential ungleich Null besitzen oder die einen Puls von einem Vorgänger erhalten haben. Für eine solche Optimierung muss eine Liste geführt werden, welche die Menge dieser Neuronen erfasst.

In Zeitscheibensimulation von [Wolff2001], basierend auf [FraHar1995], [FraBilHar1995], [Frank1997], [HarFraSch1997] und [FraHarJah1999], wird sowohl eine Liste aller Neuronen geführt, die ein Summenpotential ungleich Null besitzen (Abklingliste) als auch eine Liste, in der alle Neuronen aufgeführt sind, welche im letzten Zeitschritt gefeuert haben (Spikeliste). Der Vorteil eines solchen Ansatzes liegt darin, dass weiterhin im Kern eine – sehr einfach beschreibbare und implementierbare – zeitgetriebene Simulation verwendet werden kann. Dies wird jedoch dadurch erkauft, dass Rechenzeit in das Abklingen einer nicht unbeträchtlichen Anzahl Neuronen investiert werden muss. Gelingt es hingegen, die Berechnungen auf die Zeitpunkte zu beschränken, an denen die Neuronen Pulse von ihren Vorgängern erhalten, so kann auf eine solche Abklingphase verzichtet werden. Dies erfordert jedoch eine andere Implementierung der Neuronenmodelle, da im Falle eines eintreffenden Pulses nicht auf die Potentialwerte des letzten Zeitschritts zurückgegriffen werden kann. Vielmehr müssen aufgrund des Zeitpunkts und der Ergebnisse der zuletzt durchgeführten Berechnung die Auswirkungen des aktuellen Pulses bestimmt werden.

Es scheint daher nahe zu liegen, dass die geringe Aktivität des Netzwerks vor allem dann ausgenutzt werden kann, wenn nur die von den Neuronen ausgehenden Pulse als Ereignisse in einer ereignisgetrieben Simulation berücksichtigt werden. Da jedoch individuelle Verzögerungen zwischen sendenden Neuronen und den einzelnen Endigungen der Axone existieren, lässt sich das früheste Ereignis im System nicht alleine über die Feuerzeitpunkte der Neuronen bestimmen, sondern es müssen die individuell verzögerten Ereignisse an den Eingängen dafür herangezogen werden. Dieser Zusammenhang führt letztlich dazu, dass es günstiger ist, die individuell verzögerten Pulse als Ereignisse zu modellieren und damit zwar eine größere Menge Ereignisse zu verwalten, aber den Aufwand zur Bestimmung des frühesten Ereignisses im System zu minimieren. In [Wolff2001] kann hingegen nur eine einheitliche Verzögerung für alle Endigungen eines Axons verwendet werden. Dieser Ansatz verhindert einerseits die exakte Simulation unterschiedlicher Verzögerungspfade und des daraus resultierenden Systemverhaltens, andererseits verbaut es die individuelle Anpassung der Verzögerungen durch ein entsprechendes Lernverfahren. Diese Effekte sind jedoch gerade im Hinblick auf die Untersuchung von Synchronisationseffekten wesentlich.

3.1. Grundlegende Verfahren

3.1.1. Sequentielle Simulation

Bei einer sequentiellen Simulation werden die Zustände aller Neuronen mit Hilfe eines sequentiellen Programms berechnet. Dies bedingt automatisch eine zentrale Synchronisation der Simulation. Da es sich bei den hier betrachten Simulationen um digitale Simulation pulsverarbeitender neuronaler Netze handelt, sind sowohl die Zeit als auch die Zustandsvariablen als diskrete Werte abgebildet.

3.1.1.1. Zeitgetriebene Simulation

Die meisten Simulationssysteme für pulsverarbeitende Netze verwenden ein sequentielles, zeitgetriebenes Verfahren. Dabei werden die Zustände der Neuronen eines Netzwerks auf einem Rechnersystem in festen Zeitabständen ausgewertet. Die Zeitabstände, welche auch Zeitscheiben genannt werden, bestimmen die zeitliche Auflösung, mit der Zustandsänderungen im System beobachtet werden können. Das Verfahren bedingt, dass jedes Neuron im System in jeder Zeitscheibe berechnet oder zumindest überprüft werden muss. Die Simulation wird also durch die Zeitscheibe synchronisiert. Eine Überprüfung findet auch statt, wenn an einem Neuron keine Pulse anliegen, das heißt also auch keine erneute Berechnung des Neuronenzustands nötig wäre – davon ausgehend, dass alleine die mögliche Entstehung eines Aktionspotentials Grund für eine erneute Berechnung des Neuronenzustandes ist.

Es liegt auf der Hand, dass bei dieser Simulationsmethode die Veränderungen der inneren Zustände eines jeden einzelnen Neurons leicht zu dokumentieren sind, da diese Informationen in jeder Zeitscheibe gewonnen werden. Andererseits kann mit diesem Verfahren weder ein Vorteil aus der durchweg geringen Aktivität pulsverarbeitender neuronaler Netze gezogen werden noch aus der spärlichen Vernetzung. Vielmehr steigt die Laufzeit zur Berechnung eines Simulationsschritts mit der Zahl der Neuronen bzw. mit der Zahl ihrer Eingänge weitgehend linear an und ist ebenso weitgehend unabhängig von der Aktivität im Netzwerk. Zudem besteht auch eine lineare Abhängigkeit von der

Genauigkeit, mit der simuliert wird, da bei einer Verkürzung der Zeitscheiben entsprechend mehr Zeitscheiben pro Sekunde ausgewertet werden müssen.

Die zeitgetriebene Simulation kann durch verschiedene Mechanismen beschleunigt werden, mit dem Ziel, die Anzahl der Berechnungen pro Zeitschritt so weit wie möglich zu reduzieren. Da bei solchen Optimierungen mitunter Ereignislisten eingesetzt werden, um zum Beispiel die zur Zeit aktiven Neuronen zu speichern, ist es notwendig zu definieren, wann es sich noch um eine Zeitscheibensimulation und wann um eine ereignisgetriebene Simulation handelt. Laut Fujimotos Definition [Fujimoto2000] handelt es sich solange noch um eine zeitgetriebene Simulation wie das Fortschreiten der Zeit grundsätzlich in äquidistanten Zeitschritten erfolgt. Die Weite der Zeitschritte legt die zeitliche Auflösung fest, mit der das System simuliert wird. Der minimale Aufwand für jeden Zeitschritt einer solchen Simulation besteht aus der Überprüfung, ob irgendeine Berechnung anzustellen ist, die notwendigen Berechnungen durchzuführen und die Simulationszeit zu inkrementieren.

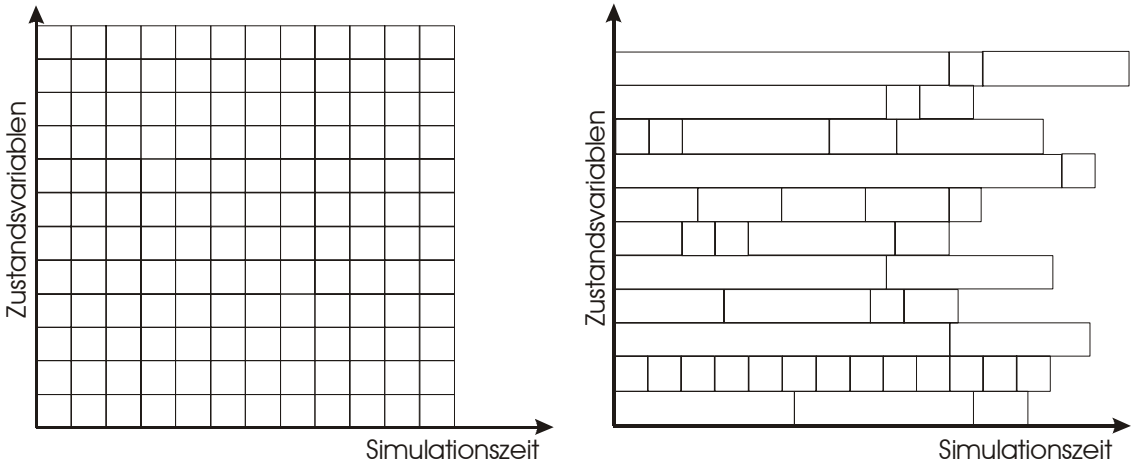


Abbildung 17: Raum-Zeit Diagramm für eine (a) zeitgetriebene und (b) ereignisgetriebene zeitdiskrete Simulation (nach [Fujimoto2000])

3.1.1.2. Ereignisgetriebene Simulation

Bei der ereignisgetriebenen sequentiellen Simulation synchronisiert eine zentrale Ereignisliste die Simulation. Sie sortiert die im System auftretenden Ereignisse in zeitlich aufsteigender Reihenfolge. Aus dieser Liste wird jeweils das früheste Ereignis entnommen und es werden die Auswirkungen dieses Ereignisses berechnet.

Ereignisse entstehen an den Zeitpunkten, zu denen wesentliche[15] Veränderungen im System stattfinden. An diesen Punkten wird der zukünftige Verlauf der Simulation bestimmt und die zeitlich nächsten Ereignisse werden berechnet, die wesentliche Veränderungen des Systems beschreiben. In

Abbildung 17 wird im Vergleich zur zeitgetriebenen Simulation verdeutlicht, dass die Zustandsvariablen des zu simulierenden Systems daher nicht in äquidistanten Abständen aktualisiert werden. Eine besondere Bedeutung kommt daher der Modellierung der Ereignisse und der sie verarbeitenden Prozesse zu, den so genannten logischen Prozessen, um tatsächlich die für die Simulation des betrachteten Systems wesentlichen

[15] Fujimoto [Fujimoto2000] spricht in diesem Zusammenhang von Zeitpunkten, zu denen „etwas Interessantes“ im System geschieht.

Zustandsänderungen erfassen zu können. Durch die Modellierung muss zudem sichergestellt werden, dass einerseits die Simulation effizient durchführbar ist und andererseits eine ausreichende Flexibilität für Anpassungen der zu verarbeitenden Prozesse vorhanden ist. Der Flexibilität sind insbesondere durch die Notwenigkeit, wesentliche Zustandsänderungen definieren zu müssen, und durch die Forderung nach einer effizienten Simulationsausführung Grenzen gesetzt.

Eine geeignete Modellierung vorausgesetzt, kann die geringe Aktivität und die spärliche Vernetzung pulsverarbeitender neuronaler Netze dazu genutzt werden, die Simulationszeit zu verkürzen, wobei die zeitliche Auflösung der Simulation nahezu beliebig genau sein kann. Dies gelingt, wenn die Anzahl der Ereignisse in direktem Zusammenhang mit der Aktivität im System steht und mit dieser entsprechend skaliert. Mit einer höheren zeitlichen Auflösung vergrößert sich die Rechenzeit geringfügig, da die Ereignisse im System genauer berechnet werden müssen. Der durch die genauere Berechnung bedingte Anstieg der Rechenzeit stellt jedoch nur einen Bruchteil des Anstiegs dar, der durch eine Verkleinerung des Zeitscheibenabstands in einer Zeitscheibensimulation verursacht würde. Dies gilt insbesondere für eine große Anzahl zu simulierender Neuronen.

Der wesentliche Unterschied zur zeitgetriebenen Simulation besteht darin, dass die Zeit während der Simulation nicht in äquidistanten Schritten fortschreiten muss, sondern von einer wesentlichen Zustandsänderung zur nächsten springt. Solche Sprünge können sich über mehrere Einheiten der gewählten diskreten Zeitauflösung erstrecken. Daher ergibt sich bei der ereignisgetriebenen Simulation systembedingt kein direkter Zusammenhang zwischen der Zeitauflösung und dem Berechnungsaufwand. Das Fortschreiten der Simulationszeit wird also von den Ereignissen bestimmt und nicht durch eine Schleife, welche die Simulationszeit in äquidistanten Schritten inkrementiert.

Da die Modellierung der Ereignisse und Prozesse in einer ereignisgetriebenen Simulation einen wesentlichen Aspekt darstellt, wird im Folgenden auf diese beiden Punkte genauer eingegangen.

3.1.1.2.1. Ereignisse

Um eine Trennung der Simulationsverwaltung und der eigentlichen Ereignisverarbeitung zu erreichen, ist es notwendig, die Ereignisse im System so zu modellieren, dass unterschiedliche Neuronenmodelle implementiert werden können, ohne dass Anpassungen am Simulationskern oder den Ereignissen notwendig werden. Zieht man für diesen Zweck die im vorangegangen Kapitel dargestellten Eigenschaften natürlicher neuronaler Netze heran, so bietet es sich an, den Entstehungszeitpunkt eines Aktionspotentials als Ereignis zu modellieren und die Neuronen als logische Prozesse, die diese Ereignisse verarbeiten. Dieser Modellierungsansatz fußt vor allem auf der Erkenntnis, dass ein Aktionspotential, welches an einer Synapse beobachtet werden kann, stets die gleiche Form aufweist. Damit wird die entscheidende Information durch den Zeitpunkt, zu dem das Aktionspotential auftritt, repräsentiert und nicht durch seine Form. Das Ereignis im System ist also alleine durch den Zeitstempel, den es trägt, ausreichend charakterisiert. Lediglich um Pulse, die zwischen den Neuronen ausgetauscht werden, und Kontrollereignisse zur Steuerung des Simulationsablaufs gegeneinander abzugrenzen, muss das Ereignis zusätzlich durch seinen Typ identifizierbar sein.

Abhängig von der gewählten Verwaltung der Ereignisse muss ein Ereignis entweder eine eindeutige Zuordnung zum Ausgang des Quellneurons oder zu den Eingängen an

den verarbeitenden Zielneuronen mit sich tragen. Wird ein Verweis auf den Ausgang des Quellneurons im Ereignis gespeichert, so handelt es sich um eine Empfängerverteilung und es muss nur das Ausgangsereignis in der Ereignisliste abgelegt werden.

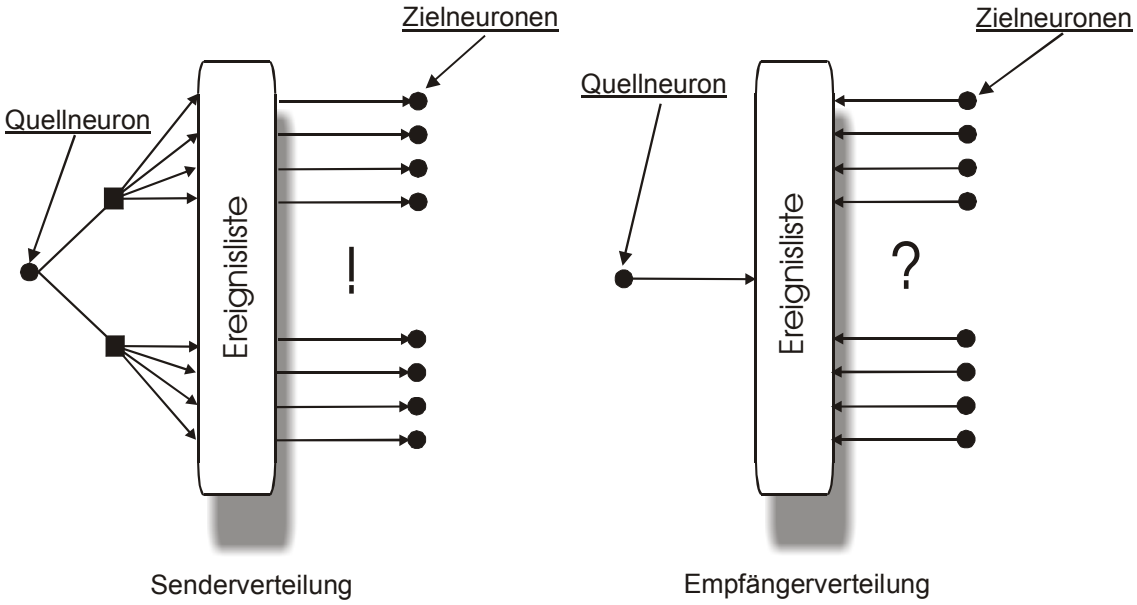


Abbildung 18: Senderorientierte und empfängerorientierte Ereignisverteilung

Bei der Entnahme des Ereignisses muss jedoch für jedes Zielneuron geprüft werden, ob es potentiell durch das aktuelle Ereignis betroffen ist – die Verteilung des Ausgangsereignisses auf die Zielneuronen findet auf der Empfängerseite statt. Zudem wird die Bestimmung des frühesten Ereignisses im System erschwert, da unterschiedliche Verzögerungen zu den Zielneuronen vorliegen können. Bei der Senderverteilung ist hingegen die Zahl Ereignisse höher, die in die Ereignisliste abzulegen sind, jedoch ist das früheste Ereignis sehr leicht zu bestimmen, wenn die Ereignisse grundsätzlich verzögert und mit einem Verweis auf den Eingang am betreffenden Zielneuron eingetragen werden (→ Abschnitt 3.2.5) – die Verteilung des Ausgangsereignisses findet in diesem Falle bereits auf der Senderseite statt. Derart ausgestattete Ereignisse muss der Simulationskern lediglich in zeitlich aufsteigender Reihenfolge in einer Ereignisliste verwalten und sie aus dieser der Reihe nach entnehmen, um sie von den Zielneuronen, auf die im Ereignis selbst verwiesen wird, verarbeiten zu lassen. Die Zielneuronen wiederum generieren aufgrund eingehender Ereignisse unter Umständen wieder ausgehende Ereignisse, die direkt in die Ereignisliste eingespeist werden. Zusätzlich kann dem Ereignis ein so genannter *look ahead* mitgegeben werden, aus dem der Empfänger ablesen kann, dass von dem Sender[16] des Ereignisses bis zu dem Zeitpunkt, der sich aus dem Zeitstempel plus *look ahead* ergibt, keine weiteren Ereignisse mehr zu erwarten sind.

Tabelle 1 zeigt den Aufbau eines Ereignisses, welches den vorangegangenen Forderungen genügt und gibt die im später vorgestellten Simulator verwendete Repräsentation der Datenfelder wieder.

[16] Da das Ereignis keine Information über den Sender trägt, muss diese Zuordnung durch den im Ereignis angegebenen Eingang erfolgen.

Zeitstempel	Zeitrepräsentation – 32Bit (unsigned long)
Zielneuron / Prozess	Neuronenadresse – 32Bit (Speicheradresse)
Zieleingang	Eingangsnummer – 32Bit (unsigned long)
look ahead	Zeitwert – 32Bit (unsigned long)

Tabelle 1: Datenfelder eines Ereignisses

3.1.1.2.2. Logische Prozesse

Die Neuronen bzw. logischen Prozesse in der Simulation können ein beliebiges Modellverhalten implementieren, wobei lediglich zwei wesentliche Voraussetzungen erfüllt werden müssen:

1. Der logische Prozess darf nur Ereignisse generieren, die entweder gleichzeitig mit dem zuletzt eingegangenen Ereignis stattfinden oder später, damit die zeitliche Konsistenz der Simulation gewahrt bleibt.
2. Der logische Prozess muss stets den frühesten Zeitpunkt bestimmen, zu dem er potentiell ein ausgehendes Ereignis erzeugen wird, auch wenn dieses Ereignis durch weitere eingehende Ereignisse revidiert werden könnte. Kann der Prozess hingegen sicher feststellen, dass aufgrund der aktuellen Eingangsereignisse und des inneren Zustands keine ausgehenden Ereignisse entstehen werden, so braucht auch nichts in die Ereignisliste eingespeist zu werden.

Erhält ein logischer Prozess keine eingehenden Ereignisse, so werden für den betreffenden Prozess auch keine Berechnungen angestellt. Es besteht jedoch jederzeit die Möglichkeit, dass ein Prozess ein Ereignis erzeugt, welches er an sich selbst versendet, um wesentliche Veränderungen innerer Zustände, die unabhängig von äußeren Einwirkungen stattfinden, zum korrekten Zeitpunkt vorzunehmen..

Aufgrund der beschriebenen Randbedingungen bietet es sich an, verwaltungstechnische Aufgaben des logischen Prozesses in einem Objekt zu konzentrieren und algorithmische Aufgaben in einem anderen. Zu den verwaltungstechnischen Aufgaben gehört einerseits die zeitliche Konsistenz für ausgehende Ereignisse sicherzustellen, und andererseits, einen *look ahead* zu berechnen, welcher aufgrund der eingehenden Ereignisse zu bestimmen ist.

Die algorithmischen Aufgaben umfassen alle Berechnungsschritte, die ausgeführt werden müssen, um den nächstmöglichen Zeitpunkt der Aktivität des betreffenden Neurons zu bestimmen, oder festzustellen, dass es unter keinen Umständen aktiv wird. Sofern als Ergebnis dieser Berechnungen ein Ereignis generiert wird, ist es die Aufgabe des verwaltenden Objekts, zu überprüfen, ob dieses Ereignis im Sinne der zeitlichen Konsistenz sicher ist. Das Ereignis ist sicher, wenn es entweder gleichzeitig mit dem zuvor eingegangenen Ereignis oder noch innerhalb des aktuellen *look ahead* stattfindet. Liegt weder der eine noch der andere Fall vor, so ist ein Kontrollereignis vom verwaltenden Objekt zu generierten, welches mit dem Zeitstempel des berechneten Ereignisses und an dessen Stelle in die Ereignisliste eingespeist wird. Treffen bei dem betreffenden Prozess, bevor er dieses „Erinnerungsereignis“ erreicht, keine weiteren Ereignisse ein, die eine erneute Berechnung im algorithmischen Objekt auslösen und damit potentiell zu neuen Ausgangsereignissen führen, so wird das zuvor berechnete und zurückgehaltene Ereignis durch das verwaltende Objekt in die Ereignisliste eingespeist.

3.1.1.2.3. Ereignisverteilung

Ereignisse geben die Zeitpunkte auftretender Aktionspotentiale an. Sie können innerhalb der Simulation „senderorientiert“ oder „empfängerorientiert“ verteilt werden. Bei der empfängerorientierten Verteilung wird das Ereignis des sendenden Neurons abgespeichert und die Empfänger überprüfen jeweils, ob für sie ein neues Ereignis vorliegt. Bei der senderorientierten Verteilung werden für alle Nachfolger Kopien des ausgehenden Ereignisses eines sendenden Neurons erzeugt und in der Ereignisliste abgelegt. Bei diesem Verfahren genügt es, das früheste Ereignis aus der Liste zu entnehmen, da in diesem Falle die Empfänger nicht explizit überprüfen müssen, ob für sie ein Ereignis vorliegt.

Die empfängerorientierte Verteilung führt letztlich zu einer ähnlichen Situation wie bei einer Zeitscheibensimulation, da zumindest für jedes Ereignis im System alle Neuronen überprüfen müssen, ob sie dieses Ereignis betrifft. Die Anzahl der Aufrufe hängt jedoch von der Aktivität im System und nicht von der zugrunde liegenden Zeitauflösung ab. Der Vorteil gegenüber der senderorientierten Verteilung der Ereignisse liegt in der weitaus geringeren Zahl der Ereignisse, die in die Ereignisliste gelangen, da für ein feuerndes Neuron nur ein Ereignis in der Liste abgelegt wird. Bei der senderorientierten Verteilung hingegen hängt die Zahl der zu speichernden Ereignisse von der Zahl der Nachfolger ab. Dadurch wird jedoch erreicht, dass die Neuronen im System immer nur dann aufgerufen werden, wenn ein Ereignis aus der Liste entnommen wird, das an das betreffende Neuron adressiert ist.

Die Anzahl der Aufrufe kann für eine empfängerorientierte Verteilung der Ereignisse grob durch $\varnothing_{Aktive\ Neuronen} * Anzahl\ Neuronen$ approximiert werden und für eine senderorientierte Verteilung mit $\varnothing_{Aktive\ Neuronen} * \varnothing_{Anzahl\ Verbindungen}$.

Da in natürlichen neuronalen Netzen die durchschnittliche Anzahl der Verbindungen eines Neurons zu seinen Nachfolgern klein gegenüber der Anzahl der Neuronen im gesamten System ist, bietet sich zur Simulation solcher Netzwerke die senderorientierte Verteilung an. Setzt man die Zahlen für den cerebralen Cortex an, welcher die bei weitem dichteste Vernetzung im Verhältnis zur Zahl der Neuronen enthält, so liegt in diesem Fall ein Verhältnis von $\sim 1.5*10^{10}$ Neuronen zu $\sim 10^5$ ausgehenden Verbindungen pro Neuron vor. Dies würde im Falle einer empfängerorientierten Verteilung der Ereignisse zu 10^5 mal mehr Aufrufen führen.

Für technische Modelle natürlicher neuronaler Netze gilt dies nicht unbedingt. So entspricht die durchschnittliche Zahl der Verbindungen eines Neurons zu seinen Nachfolgern in einem vollvernetzten Netzwerk der Anzahl der Neuronen in demselben, d.h. in einem solchen Falle entstehen für beide Verteilungsverfahren die gleiche Zahl Aufrufe. Schichtweise voll vernetzte Netzwerke haben ein ähnlich ungünstiges Verhältnis. Für ein dreischichtiges Netzwerk mit jeweils n Neuronen pro Schicht beträgt die Zahl der Neuronen 3n und die Zahl der ausgehenden Verbindungen eines Neurons $\frac{2n}{3}$. Damit ergeben sich mit a aktiven Neuronen für eine empfängerorientierte Verteilung $a \cdot 3n$ Aufrufe und für die senderorientierte Verteilung $a \cdot \frac{2n}{3}$ Aufrufe. Deshalb fallen für eine empfängerorientierte Verteilung lediglich $\frac{9}{2}$ mal mehr Aufrufe an und der Vorteil einer senderorientierten Verteilung tritt nicht so deutlich zu Tage wie im Falle der Simulation natürlicher neuronalen Netze.

Da jedoch vollvernetzte Netzwerke gewissermaßen die ungünstigste Situation für eine derartige Simulation beschreiben und selbst in diesem Falle eine empfängerorientierte Verteilung lediglich genauso viele Aufrufe wie eine senderorientierte Verteilung benötigt und im Regelfall die Zahl der Aufrufe für eine senderorientierte Verteilung um Faktoren geringer ausfällt, ist für eine ereignisgetriebene Simulation von pulsverarbeitenden neuronalen Netzen eine senderorientierte Verteilung vorzuziehen. Diese Argumentation hält, wie ausreichend Speicher für die große Zahl Ereignisse verfügbar ist und eine Ereignisliste zum Einsatz kommt, deren Zugriffszeiten weitgehend unabhängig von ihrer Füllung sind. Außerdem bezieht sich diese Abschätzung ausschließlich auf eine ereignisbasierte Kommunikation und Berechnung. Werden hingegen, zum Beispiel durch Lernverfahren, neuronenübergreifende Informationen benötigt, so können durch derartige Anforderungen enorme Ereigniszahlen entstehen und den Rückgriff auf eine empfängerorientierte Verteilung sinnvoll machen.

3.1.1.2.4. Ereignisliste

Die Ereignisliste ist der zentrale Umschlagplatz innerhalb der ereignisgetriebenen Simulation. Hier werden die Ereignisse in zeitlich aufsteigender Reihenfolge sortiert und verwaltet. Vorne in der Liste steht das früheste Ereignis, das im System existiert. Die Sortierung und die Forderung nach zeitlicher Konsistenz der Simulation bedingt, dass das erste Ereignis in der Liste sicher verarbeitet werden kann, da die Auswirkungen dieser Verarbeitung nur gleichzeitige oder spätere Ereignisse erzeugen.

Ereignislisten, die für die ereignisgetriebene Simulation pulsverarbeitender neuronaler Netze geeignet sind, müssen vor allem auch noch bei einem sehr hohen Ereignisaufkommen effizient arbeiten (→ Ereignisverteilung). Wie in dem folgenden Abschnitt zur Auswahl einer geeigneten Ereignisliste zu entnehmen ist, eignen sich hierfür nur wenige Algorithmen. Ein hohes Ereignisaufkommen kann einerseits zeitweilig entstehen, da lediglich die durchschnittliche Aktivität gering ist, eine momentane Aktivität jedoch recht hoch sein kann (→ z. B. durch Neuronen, die einen so genannten *burst* aussenden), andererseits können viele Ereignisse durch ein entsprechend großes Netzwerk verursacht werden, das trotz einer geringen Aktivität durch die absolute Menge der Neuronen sehr viele Ereignisse in die Ereignisliste einspeist.

Im Abschnitt zur Ereignisverteilung wurde bereits deutlich gemacht, dass eine senderorientierte Verteilung der Ereignisse, bei einem ausreichenden Speicherausbau, die günstigere Alternative für die Simulation von pulsverarbeitenden neuronalen Netzen darstellt. Die senderorientierte Verteilung führt jedoch zu einer großen Zahl von Ereignissen, die durch die Ereignisliste verwaltet werden müssen. Den Ausführungen von [RönAya1997] ist zu entnehmen, dass sich die Algorithmen für Prioritätswarteschlangen unterschiedlich in Abhängigkeit ihres Füllstandes und des Zugriffprofils verhalten. Viele der Algorithmen zeigen einen deutlichen Leistungseinbruch, wenn die Größe der Ereignisliste über 10000 Elemente steigt. Lediglich die Lazy Queue und die Calendar Queue verhalten sich für große Füllstände gutmütiger. Die Eignung der verschiedenen Algorithmen hängt jedoch auch von der Verteilung der einzufügenden Daten ab und dem Einfüge- und Entnahmeverhalten. Dadurch hängt die Wahl eines geeigneten Algorithmus von den Eigenschaften der Simulation ab. Die erwarteten und die ungünstigsten Komplexitäten der Einfüge- und Entnahmeoperationen von Calendar Queue und Lazy Queue weisen eine enorme Spanne auf - zwischen O(1) und O(n). Daher ist gerade bei diesen Algorithmen eine starke Abhängigkeit von den genannten Parametern zu erwarten. Sofern es jedoch gelingt, die zugrunde liegenden Datenstrukturen so auszulegen, dass eine

Restrukturierung der Datenstruktur während der Simulation auszuschließen ist, so sind die Algorithmen anderen deutlich überlegen[17].

Da eine derartige Auslegung der Datenstrukturen für die Simulation pulsverarbeitender neuronaler Netze möglich ist und da die Ergebnisse aus [RönAya1997] dafür sprechen, wurde sowohl eine auf einem binären Baum basierende Implementierung aus der Standard Template Library (STL) als auch die Calendar Queue für den Simulator implementiert. Darüber hinaus wurde für den Hardwarebeschleuniger ein weiterer Algorithmus verwendet, der so genannte Fishspear-Algorithmus [FisPat1994], der insbesondere für langsamere Speicher geeignet ist. Die folgenden Grafiken sind in Rahmen von Messungen[18] in [Werkhausen2002] entstanden und zeigen den Geschwindigkeitsvorteil der Calendar Queue gegenüber der Priority Queue (STL) für hohe Füllstände der Warteschlange. Damit bestätigen diese Messungen die Ergebnisse von [RönAya1997].

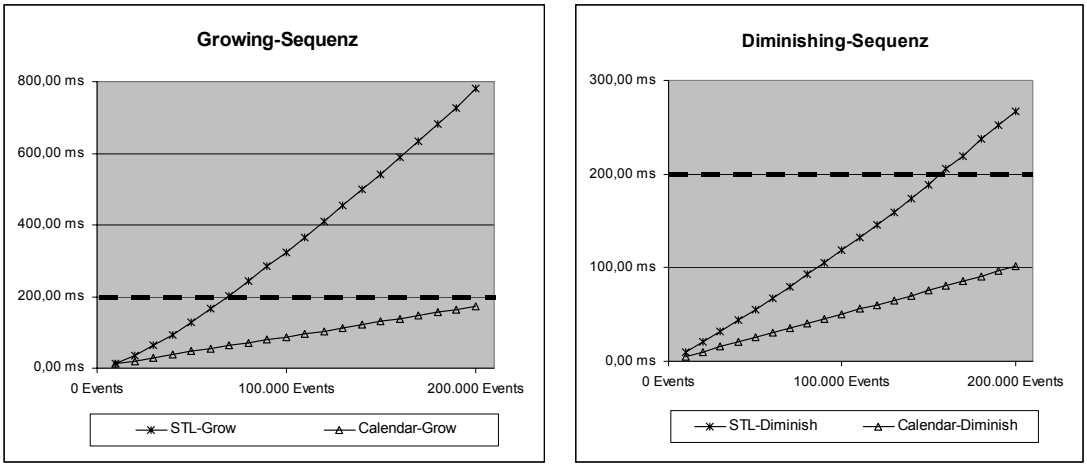


Abbildung 19: Vergleich der STL Priority und der Calendar Queue für Füllstände bis 200.000 Elemente.

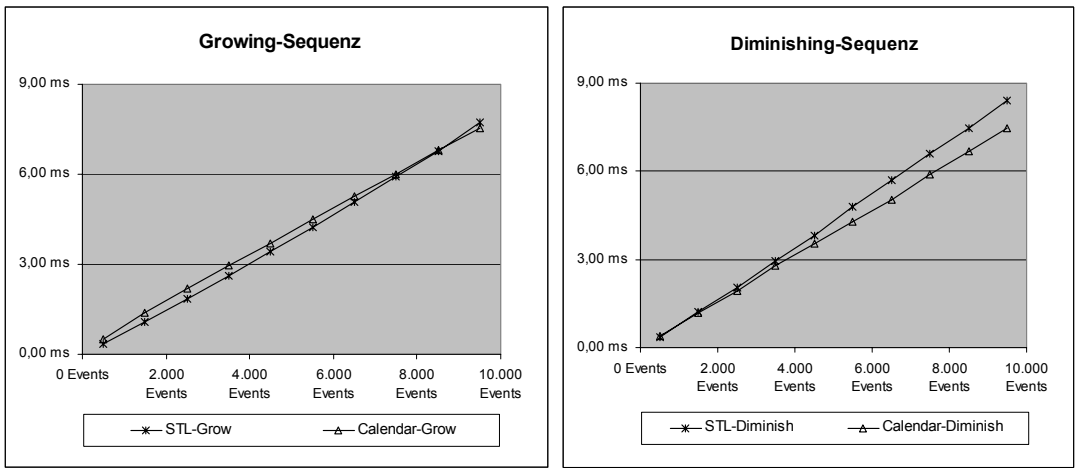


Abbildung 20: Vergleich der STL Priority und der Calendar Queue für Füllstände bis 10.000 Elemente.

[17] Die dynamische Anpassung der Datenstrukturen wird in [Marin1997] als hauptsächlicher Nachteil dieser Algorithmen herausgestellt, wobei jedoch in [Marin1997] auch nur vergleichende Messungen mit bis 10.000 Ereignissen vorgenommen wurden.

[18] Die Messungen wurden auf einem PC mit dem Betriebssystem Windows NT 4.0, einem auf 1066 MHz getakteten Celeron-Prozessor (133Mhz Frontbus) und 768MB Arbeitsspeicher ausgeführt.

Ab einem Füllstand von ca. 10000 Elementen zeigt die Calendar Queue deutliche Leistungsvorteile gegenüber der STL-Implementierung, wobei diese besonders stark bei der so genannten *Growing-Sequenz* zu Tage treten und weniger bei der *Diminishing-Sequenz*.

Die unterschiedlichen Ergebnisse begründen sich durch das unterschiedliche Einfüge- und Entnahmeverhalten der Testsequenzen. Bei der *Growing-Sequenz* werden im Wechsel 30 Elemente in die Warteschlange eingefügt und 20 entnommen, bis eine zuvor festgelegte Größe der Warteschlange erreicht ist. In der *Dimninishingsequenz* wird die Warteschlange auf eine festgelegte Größe mit Elementen aufgefüllt und dann werden im Wechsel 30 Elemente entnommen und 20 eingefügt, bis die Warteschlange entleert ist. Daher dominieren bei der Growing-Sequenz die Einfügeoperationen und bei der Diminishing-Sequenz die Entnahmeoperationen. Deutlich ist zu erkennen, dass bei der auf einem binären Baum basierenden STL Priority Queue die Einfügeoperation erheblich zeitintensiver ist als die Entnahmeoperation und dies insbesondere bei hohen Füllständen der Warteschlange zum Tragen kommt.

Beide Implementierungen können wahlweise in dem später vorgestellten Simulator aktiviert werden. Es bietet sich jedoch durchweg die Verwendung der Calendar Queue aufgrund der kürzeren Zugriffszeit an. Der Calendar Queue Algorithmus wird im Folgenden detaillierter beschrieben.

Calendar Queue Algorithmus

Für den von Randy Brown [Brown1988] vorgestellten Algorithmus für Prioritätswarteschlangen wurde experimentell eine mittlere Laufzeit von O(1) je *hold*-Operation[19] nachgewiesen. Aufgrund der einfachen Struktur und des geringen Overheads ist die Calendar Queue sowohl für kleinere als auch größere Mengen von Ereignissen effizient einsetzbar.

Die Calendar Queue verwendet eine kalenderartige Struktur zum Verwalten der Ereignisse. Dieser Kalender ist dabei in eine festgelegte Anzahl von Monaten gleicher Größe eingeteilt. Jeder Monat des Kalenders besteht aus einem Behälter (*bucket*), welcher die Ereignisse dieses Jahresabschnitts aufnimmt. Alle Ereignisse eines Behälters finden im gleichen Zeitintervall eines Jahres statt, stammen aber eventuell aus unterschiedlichen Jahren und sind untereinander nicht sortiert. Als Beispiel ist in Abbildung 21 ein Kalender mit 4 Monaten von jeweils 25 Tagen dargestellt.

Ein Ereignis zum Zeitpunkt t wird dem *bucket* b aus einem Kalender mit n Monaten, der Monatslänge m über

$$b = \frac{t \bmod (n \cdot m)}{m} \qquad \text{Gl. (21)}$$

zugeordnet. Eine Zuordnung zu einem *bucket* findet demnach in konstanter Zeit mit Hilfe einer Division und einer Modulo-Operation statt.

Zur Bestimmung des frühesten Ereignisses muss der erste nicht leere *bucket* gefunden werden und daraus das Ereignis mit dem kleinsten Zeitstempel bestimmt werden. Da die Ereignisse in einem *bucket* unsortiert abgelegt sind und außerdem zwischen Ereignissen

[19] Eine *hold*-Operation entspricht dem Entfernen und Einfügen von je einem Ereignis, d.h. die Größe der Queue bleibt unverändert.

aus dem aktuellen und späteren Jahren, die sich im gleichen *bucket* befinden können, unterschieden werden muss, ist hierfür eine entsprechende Suche zu implementieren.

Monat 1 Bucket 0 [Tag 0..24] (t mod 100) div 25 = 0	Monat 2 Bucket 1 [Tag 25..49] (t mod 100) div 25 = 1	Monat 3 Bucket 2 [Tag 50..74] (t mod 100) div 25 = 2	Monat 4 Bucket 3 [Tag 75..99] (t mod 100) div 25 = 4
0 23 101 17 512	244 27 1344		79 77 77 75 99 99

Abbildung 21: Calendar Queue Struktur mit 4 Monaten und einer Monatsbreite von 25 Tagen. Ein Kalenderjahr dauert in diesem Kalender also 100 Tage.

Um den Zeitaufwand t^h für eine *hold*-Operation auf der Calendar Queue abschätzen zu können, wird von unendlich vielen *buckets* ausgegangen, also nur einem Kalenderjahr. Vor dem *bucket*, in dem sich das gesuchte Ereignis zusammen mit *n-1* weiteren Ereignissen befindet, seien *m* leere *buckets*. Mit der Zeit t^{d} zur Berechnung eines neuen Ereignisses inklusive dem Einfügen desselben in den Kalender, der Zeit t^b zum Durchsuchen eines leeren *buckets* und der Zeit t^c pro Ereignis zur Suche innerhalb eines *buckets* folgt:

$$t^h = t^d + m \cdot t^b + n \cdot t^c \qquad \text{Gl. (22)}$$

Kann für die Differenz zwischen den Zeitstempeln eines neu in die Ereignisliste einzufügenden Ereignisses und dem zuletzt entfernten Ereignis ein Maximalwert angegeben werden, dann kann die Anzahl der *buckets* ohne Leistungseinbußen so weit reduziert werden, dass ein Jahr des Kalenders gerade diesen maximalen Abstand überdeckt [EriLadLaM1994]. Ebenso lässt sich nach [EriLadLaM1994] mit der durchschnittlichen Differenz μ („durchschnittliche Sprungweite“) von dem Zeitstempel des zuletzt aus der Warteschlange entfernten Ereignisses zu dem Zeitstempel des neu generierten Ereignisses die optimale Breite der *buckets* δ^{opt} in einem Calendar Queue mit *N* Ereignissen angeben:

$$\delta^{opt} = \sqrt{\frac{2 \cdot t^b}{t^c}} \cdot \frac{\mu}{N}. \qquad \text{Gl. (23)}$$

Für die zu erwartende durchschnittliche Zeit t^h einer *hold*-Operation gilt dann:

$$t^h = t^c + \sqrt{2 \cdot t^b \cdot t^c} + t^d. \qquad \text{Gl. (24)}$$

Aus Gleichung 23 ist ersichtlich, dass für sehr große N, d.h. eine Prioritätswarteschlange mit sehr vielen Ereignissen, δ^{opt} sehr klein wird – vorausgesetzt, μ ist nicht ungewöhnlich groß. Da genau dies bei der Simulation pulsverarbeitender neuronaler Netze aber der Fall ist, nämlich eine relativ große Menge Ereignisse bei einer kleinen bis mittleren durchschnittlichen Sprungweite μ, bietet sich weitgehend unabhängig von t^b und t^c ein *bucket* mit minimaler Breite an – im später vorgestellten Simulator entspricht die Breite daher genau einer Einheit der zugrunde liegenden Zeitauflösung. Dadurch reduziert sich t^c auf die Kosten für die Entnahme des

letzten Listenelements eines *buckets*, da nach der Bestimmung eines *buckets* keine Suche in diesem erfolgen muss. Auch die Bestimmung des *buckets* vereinfacht sich zu

$$b = t \bmod (n \cdot m). \qquad \text{Gl. (25)}$$

Für die Ausführung auf Standardprozessoren lässt sich t_d weiter reduzieren, indem für die Größe des Kalenders und die Breite des *bucket* eine Potenz von zwei gewählt wird, so dass Bitmanipulationen anstelle von Multiplikation, Division und Moduloberechnung eingesetzt werden können. Es ergibt sich dann mit $\delta_{opt} = 1$ und

$$t \bmod 2^x = t \,\text{and}\, (2^x - 1) \qquad \text{Gl. (26)}$$

$$b = t \,\text{and}\, ((n \cdot m) - 1), \qquad \text{Gl. (27)}$$

wobei $(n \cdot m) - 1$ für feste Kalendergrößen im Voraus festgelegt werden kann und daher eine Konstante ist.

Eine detailliertere Untersuchung des Calendar Queue Algorithmus und der Optimierungsmöglichkeiten im Hinblick auf die Simulation pulsverarbeitender Netze findet sich in [Werkhausen2002].

3.1.2. Verteilte Simulation

Die Darstellungen in diesem Abschnitt folgen in weiten Teilen den Ausführungen von [Fujimoto2000].

Wenn Netzwerke zu groß für die Simulation durch ein sequentiell ausgeführtes Programm sind oder Netzwerke mittlerer Größe unabhängig voneinander zu berechnende Netzabschnitte besitzen, so bietet sich die Verteilung der Simulation auf mehrere Rechner an. Einerseits wird dadurch die Simulation sehr großer Netzwerke überhaupt erst möglich und andererseits kann bereits die Simulation von Netzwerken mittlerer Größe beschleunigt werden.

Um einen Nutzen aus der verteilten Simulation ziehen zu können, muss der Gewinn durch die parallele Berechnung unterschiedlicher Netzwerkabschnitte die Kosten übertreffen, welche durch die verteilungsbedingte zusätzliche Kommunikation entstehen. Um dieses Verhältnis möglichst günstig zu gestalten, zielen die Verfahren zur Verteilung einer Simulation darauf ab, die Kommunikation so weit wie möglich zu reduzieren und durch eine geschickte Aufteilung die Rechenlast möglichst gleichmäßig auf die beteiligten Rechner zu verteilen. Besonders kritisch ist hierbei die Synchronisation der an der Simulation beteiligten Rechnersysteme. Dies trifft insbesondere für die ereignisgetriebene Simulation zu.

3.1.2.1. Zeitgetriebene Simulation

Die Verteilung der Simulation erfordert für eine Zeitscheibensimulation andere Maßnahmen als für eine ereignisgetriebene Simulation. Für eine Zeitscheibensimulation ist es notwendig, dass alle Rechner jeweils zur Zeitscheibe synchronisiert werden. Dies führt dazu, dass die Simulation erst zur nächsten Zeitscheibe übergehen kann, wenn alle Rechner die aktuelle Zeitscheibe abgearbeitet haben. Eine Vorausschau ist in einer reinen Zeitscheibensimulation nicht vorgesehen. Da die nächste Zeitscheibe mit einer Grenze zu vergleichen ist, die von allen Rechnern erreicht werden muss, wird diese Synchronisationsmethode auch *barrier synchronisation* genannt. Durch diese starr ge-

koppelte Synchronisation ist eine so verteilte Simulation sehr empfindlich gegenüber Veränderungen der Lastverteilung.

Daher werden Systeme zur Simulation pulsverarbeitender Netze, die auf verteilten Zeitscheibenverfahren basieren, häufig durch zusätzliche Mechanismen ergänzt, die es erlauben, *look aheads* für die an der Simulation beteiligten Rechner zu bestimmen. Als Beispiel bietet sich hier die parallele Implementierung des Simulators GENESIS (PGENESIS) an. Die Parallelisierung findet bei PGENESIS nicht auf Neuronenebene statt, sondern dort können Aufgaben durch die Erzeugung von Threads auf entfernte Rechner ausgelagert werden. Dies geht blockierend und nicht blockierend, wobei allein ein nicht blockierender Aufruf einen Gewinn verspricht. Zudem steht eine *barrier synchronisation* zur Verfügung, die nicht an der Zeitscheibe orientiert wird, sondern an ausgezeichneten Ereignissen innerhalb der Skripte, welche die Simulation auf den einzelnen Rechnern steuern. Eine solche *barrier synchronisation* kann entweder auf die gesamte parallele Simulation wirken, oder auf eine Zone, die eine begrenzte Zahl Rechner einschließt. Letztlich ist es Aufgabe des Benutzers, für einen blockadefreien Ablauf der Simulation zu sorgen, indem er die Simulationsskripte für alle beteiligten Rechner entsprechend sorgfältig auslegt. Dies ist im Anbetracht der vielfältigen Möglichkeiten zur Blockade einer verteilten Simulation jedoch ein schwieriges Unterfangen und führt letztlich dazu, solche Simulationen eher konservativ auszulegen und dadurch Parallelisierungspotentiale nicht auszuschöpfen.

3.1.2.2. Ereignisgetriebene Simulation

Bei einer ereignisgetriebenen Simulation sind die Rechnersysteme nicht so starr gekoppelt, da die Ereignisse, die alle Rechner untereinander austauschen, meist in größeren Abständen erzeugt werden, als in den Abständen, die durch die zugrunde liegende Zeitauflösung vorgegeben sind. Mehrere Kommunikationskanäle zwischen den Rechnern, die durch Verbindungen zu mehreren Rechnern entstehen, bedingen jedoch auch in der ereignisgetriebenen Simulation die Bereitstellung von *look aheads*. Insbesondere wenn einer der Vorgänger keine Ereignisse an einen nachfolgenden Rechner sendet, bleibt gewissermaßen die Zeit für den Nachfolger auf diesem Kommunikationskanal eingefroren. Wodurch schließlich der betreffende Rechner oder die gesamte Simulation blockiert. Eine solche Situation kann auch entstehen, wenn der Vorgänger in der Simulationszeit voranschreitet, denn es reicht, wenn alleine die Neuronen inaktiv bleiben, die Verbindung zu dem betreffenden nachfolgenden Rechner haben.

In der verteilten (*distributed*) ereignisgetriebenen (*event driven*) Simulation (*DES*) überwindet man solche Blockaden durch zwei unterschiedliche Simulationsverfahren: Das konservative und das optimistische Simulationsverfahren. Einfach dargestellt verarbeitet das konservative Verfahren lediglich „sichere“ Ereignisse, und das optimistischen Verfahren alle verfügbaren Ereignisse, auch wenn noch potentiell weitere Ereignisse eintreffen können, welche die zuvor getätigte Berechnung falsifizieren würden.

Dies bedeutet, dass beim konservativen Verfahren der nachfolgende Rechner tatsächlich nur bis zu der Zeit simulieren darf, die ihm die Ereignisse und *look aheads* von seinen Vorgängern vorgegeben. Konservative Verfahren erfordern also, dass alle Mittel ausgeschöpft werden, um einen maximalen *look ahead* bereitzustellen, und eine Strategie, um Blockaden grundsätzlich zu vermeiden bzw. diese nach ihrer Entstehung zu entdecken und aufzulösen.

Eine Blockade lässt sich vermeiden, wenn jeder Rechner Nullnachrichten an alle Nachfolger versendet, sobald er in der Simulation fortgeschritten ist. Dadurch werden Nachfolger auch dann über den Simulationsfortschritt des Vorgängers informiert, wenn aufgrund der Simulation keine Nachrichten an die Nachfolger versendet wurden. [ChaMis1982] hat nachgewiesen, dass eine Simulation bei der Verwendung dieses Verfahrens blockadefrei ist. Ein großer Nachteil dieses Verfahrens ist jedoch das hohe Kommunikationsaufkommen, da eine beträchtliche Zahl von Nullnachrichten generiert werden kann.

Mit einer Kombination aus Blockadeerkennung und –auflösung kann ein solches Kommunikationsaufkommen vermieden werden, jedoch gestaltet sich in den meisten Fällen die Blockadeerkennung schwierig. Empfängt zum Beispiel ein Rechner von dem Vorgänger, der den kleinsten *look ahead* bereitstellt, keine weiteren Nachrichten, so kann dies an einer durch die Simulation bedingten Inaktivität an dem betreffenden Ausgang des Vorgängers liegen oder an dessen Blockade. Nur im letzteren Fall liegt tatsächlich eine Blockade vor. Prinzipiell scheint diese Situation lösbar, indem an den Vorgänger eine Anfrage gestellt wird. Jedoch führt dies möglicherweise wieder zu einem ähnlich hohen Kommunikationsaufkommen wie bei der Verwendung von Nullnachrichten.

Aus den genannten Gründen bietet es sich an, die Blockadeerkennung und –auflösung mit einer Blockadevermeidung zu kombinieren, wobei die Eigenschaften der zu simulierenden Netze zur Optimierung herangezogen werden.

Beim optimistischen Verfahren müssen einerseits die wesentlichen Informationen zur Rekonstruktion früherer Zustände abgespeichert werden, um alle Berechnungen revidieren (*roll-back*) zu können, die durch neu eintreffende Ereignisse falsifiziert wurden (*out of order events*); andererseits müssen solche Ereignisse erkannt und es muss ein Mechanismus implementiert werden, der alle Nachfolger der betroffenen Neuronen über die notwendige Korrektur der Simulation in Kenntnis setzt. *Look aheads* erübrigen sich bei diesem Verfahren, da ohnehin – ungeachtet des zeitlichen Fortschritts der Vorgänger – eingehende Ereignisse verarbeitet werden.

Die Vorteile der konservativen Simulation liegen in dem geringeren Speicherbedarf und dem einfacheren Aufbau der logischen Prozesse, welche die Neuronen modellieren. Nachteilig ist die starke Abhängigkeit von den verfügbaren *look aheads* im System, insbesondere da diese stark problemabhängig sein können. Auch die Mechanismen zur Blockadevermeidung oder –erkennung und –auflösung reduzieren zusätzlich die Effizienz der Simulation. Beim optimistischen Verfahren stellen der höhere Speicherbedarf und die aufwendigere Implementierung der logischen Prozesse die wesentlichen Nachteile dar. Zudem wird die Effizienz der Simulation mit jedem *roll-back* reduziert. Der wesentliche Vorteil liegt in der besseren Ausnutzung des Parallelisierungspotentials, denn solange kein *roll-back* notwendig ist und eine ausreichend gute Lastverteilung vorliegt, wird die überwiegende Rechenzeit für die Simulation selber und nicht für Synchronisationsmechanismen aufgewendet.

3.1.2.3. Nebenläufigkeiten – Parallelisierungspotential

Die Parallelisierung einer Simulation und damit eines Programms zur Berechnung von Prozesszuständen, respektive Neuronenzuständen, kann grundsätzlich auf unterschiedlichen Ebenen erfolgen, die sich im Wesentlichen durch ihre Granularität unterscheiden:

- Taskebene (Methodenebene)

Auf der Methodenebene werden Teilaufgaben, die asynchron zu anderen ausgeführt werden können, auf entfernte Rechner ausgelagert. Damit diese Art der Parallelisierung einen Gewinn mit sich bringt, muss einerseits die ausgelagerte Teilaufgabe ausreichend rechenzeitaufwendig und andererseits die Latenz des verbindenden Netzwerks sehr gering sein. Andernfalls muss sogar mit einer Verschlechterung der Simulationseffizienz gerechnet werden, also mit einer Verlängerung der Gesamtsimulationsdauer. Diese Art der Parallelisierung wird zum Beispiel in PGENESIS verwendet. Sie bietet sich für eine ereignisgetriebene Simulation pulsverarbeitender neuronaler Netze nicht an, da die Neuronenmodelle in der Regel nicht so komplex sind, dass es sinnvoll ist, Teile ihrer Berechnung auszulagern.

- Prozessebene (logischer Prozess - Objektebene)

  Bei einer Parallelisierung auf der Ebene logischer Prozesse bzw. auf der Ebene der Objekte, die sich im Simulationssystem identifizieren lassen, werden ganze Objekte auf entfernte Rechner ausgelagert. Die Ergebnisse der einzelnen Objekte müssen durch entsprechende Kommunikationsmechanismen und Synchronisationsmechanismen den anderen Objekten der Simulation wieder zugeführt werden. Die zu simulierenden Neuronen, repräsentiert durch logische Prozesse, sind solche Objekte. Die Parallelisierung auf der Objektebene kann außerdem eine unterschiedliche Granularität besitzen. Zum Beispiel können Objekte wie die logischen Prozesse zu Gruppen zusammengefasst und im Sinne einer parallelen Ausführung als ein Objekt aufgefasst werden.

  Darüber hinaus können jedoch auch andere Objekte der Simulation ausgelagert werden. Bei der ereignisgetriebenen Simulation kommen hierfür z. B. die Ereignisliste oder Teile des Simulatorkerns in Frage. Bei der Verteilung von Objekten auf verschiedene Rechner ist es von entscheidender Bedeutung, dass die Objekte entsprechend rechenzeitaufwendige Operationen durchführen und dass die Kommunikation zwischen den Objekten möglichst gering ausfällt und eine niedrige Latenz aufweist.

  Werden die Pulse in pulsverarbeitenden neuronalen Netzen als Ereignisse der ereignisgetriebenen Simulation modelliert, bieten sich die Neuronen als zu verteilende Objekte einer verteilten ereignisgetriebenen Simulation an. Da jedoch die Berechung eines Neurons einen Standardrechner bei weitem nicht auslastet, ist es notwendig, die Neuronen zu Partitionen zu gruppieren und schließlich diese als Objekte der parallelen Simulation zu betrachten.

- Programmebene (Prozess des Betriebssystems – Prozessebene)

  Eine Parallelisierung auf Programmebene bedeutet in der Regel eine Ausführung des gleichen Programms auf verschiedenen Rechnern mit unterschiedlichen Parametern oder Anfangsbedingungen. Die Simulation wird hierbei nicht im eigentlichen Sinne parallelisiert, sondern die Ausführung einer Versuchsreihe, die mehrere, sonst sequentiell ablaufende Simulationen notwendig macht, wird parallelisiert.

  Dies ist die einfachste Form der Parallelisierung, da keinerlei Synchronisation erforderlich ist. Eine Beschleunigung komplexer in sich parallelisierbarer Simulationen ist damit jedoch nicht zu erreichen. Da es jedoch vor allem das Ziel der parallelisierten Simulation pulsverarbeitender neuronaler Netze ist, die Simulation von großen Netzwerken zu ermöglichen, spielt die Parallelisierung auf Programmebene bei den hier angestellten Betrachtungen keine Rolle.

3.1.2.3.1. Partitionierung

Bei der Verwendung von Neuronen bzw. deren Gruppierung zu Partitionen als Objekte der parallelen Simulation wird das Netzwerk durch Aufspaltung in Teilnetze auf die einzelnen Rechnerknoten verteilt (Abbildung 22). Es leuchtet ein, dass eine Verteilung erst ab einer gewissen Netzgröße einen Gewinn verspricht, da die Aufteilung des Netzwerks grundsätzlich Rechenzeit für die Kommunikation zwischen den Rechnern beansprucht.

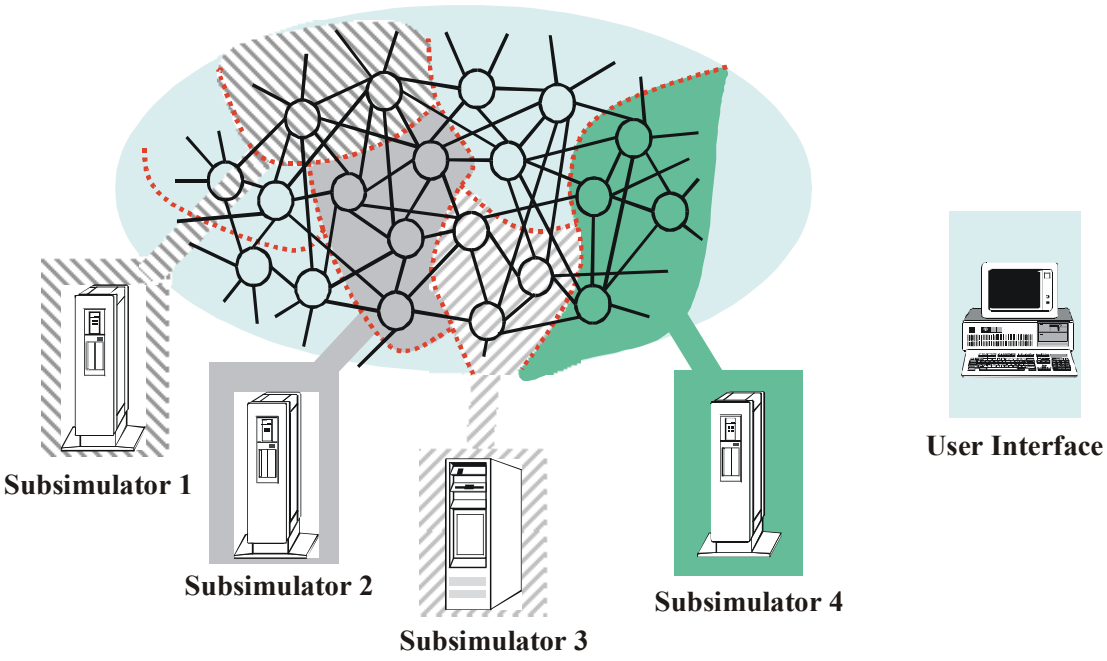


Abbildung 22: Partitionierung eines Netzwerks auf mehrere Rechner

Für die Partitionierung der Netzwerke kommen Algorithmen[20] zur Graphenpartitionierung in Betracht, wobei die Neuronen die Knoten und die axonalen Verbindungen die Kanten des Graphen bilden. Mit Hilfe der Anzahl geschnittener Verbindungen wird die Güte der Partitionierung bestimmt. Es ist dabei zu beachten, dass nicht jede durchtrennte Kante die Kommunikationskosten erhöht. Denn synaptische Verbindungen, die zwar auf unterschiedlichen Zielneuronen eines Rechners enden, aber von demselben Axon herrühren, verursachen nur einmal Kommunikationskosten, solange die Ereignisse durch den Empfänger und nicht durch den Sender auf die Zielneuronen verteilt werden (Abbildung 23). Die Verzögerung einer Verbindung kann mit in die Kostenfunktion einfließen, da in Abhängigkeit der gewählten Synchronisationsmethode diese von größeren Verzögerungswerten profitieren kann.

Um eine möglichst gute Lastverteilung durch die Partitionierung gewährleisten zu können, werden die Kosten zur Berechnung der verwendeten Neuronenmodelle herangezogen. Damit lässt sich die Lastverteilung jedoch nur unter der Annahme optimieren, dass die Aktivität gleichmäßig über das gesamte Netzwerk verteilt ist. Die Aktivität der Neuronen kann aber nicht a priori bestimmt werden, da sie von den an das Netz angelegten Stimuli abhängt und sich zudem im Verlaufe der Simulation dynamisch verändert.

---

[20] Zum Beispiel: [FidMat1982], [Gupta1996], [HenLel1992], [HenLel1993], [KarKum1998], [KarKum1998a], [KarKum1998b], [KerLin1970]

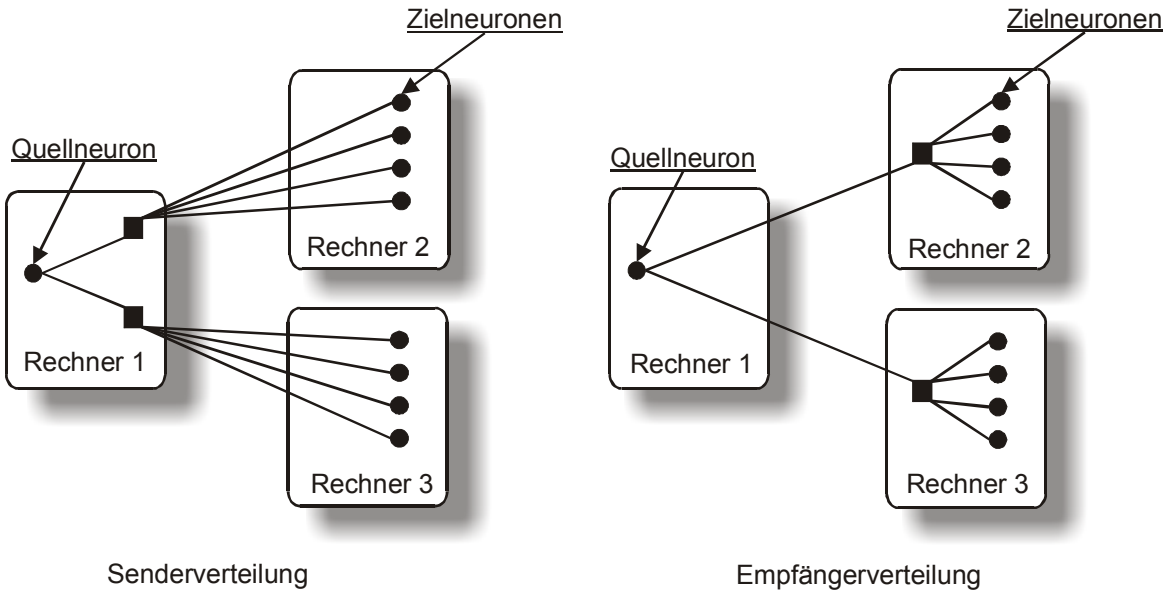


Abbildung 23: Verteilung der Ereignisse durch den Sender oder den Empfänger

Dem kann nur durch eine dynamische Anpassung der Partitionierung während der Simulation entgegengewirkt werden. Fraglich sind jedoch in diesem Zusammenhang die Kriterien, nach denen Neuronen von einer Partition in eine andere verschoben werden sollten. Hinzu kommt ein erhöhter Verwaltungsaufwand und ein deutlich gesteigertes Kommunikationsaufkommen, da einerseits Daten zur Bestimmung der Auslastung gesammelt und andererseits Neuronen von einem Rechner auf einen anderen verlagert werden müssten. Für das im folgenden Kapitel vorgestellte Simulationssystem SPIKELAB wurde deshalb auf eine dynamische Anpassung der Partitionierung verzichtet.

3.2. Ereignisgetriebene Simulation pulsverarbeitender neuronaler Netze

3.2.1. Leistungscharakteristik zeitgetriebener und ereignisgetriebener Simulation

Es sei die aufzuwendende CPU-Zeit in Bruchteilen einer Sekunde für eine Zeitscheibensimulation eines Neuronenmodells gegeben durch t_z und für eine ereignisgetriebene Simulation durch t_e[21]. Weiterhin sei a_g die gesamte Aktivität im Netzwerk, das heißt die Zahl der feuernden Neuronen pro Sekunde. Die durchschnittliche Zahl der Verbindungen zu Nachfolgern sei mit s gegeben und die Zahl der Neuronen mit n. Die Zeitauflösung werde durch t_r in Bruchteilen einer Sekunde angegeben.

Dann ergibt sich für die ereignisgetriebene Simulation ein Rechenzeitaufwand (Kosten) c_e für eine simulierte Sekunde von

$$c_e = a_g \cdot t_e \cdot s \qquad \text{Gl.(28)}$$

und c_z für die Zeitscheibensimulation entsprechend mit

[21] Der Overhead der Simulationsverfahren sei in den Zeiten t_z und t_e bereits enthalten.

$$c_z = \frac{n \cdot t_z}{t_r}. \qquad \text{Gl.(29)}$$

Bei diesen Relationen wird sowohl für die ereignisgetriebene Simulation die schwache Abhängigkeit von der gewählten Zeitauflösung vernachlässigt als auch die schwache Abhängigkeit von der Gesamtaktivität des Netzes für die Zeitscheibensimulation. Werden diese durch $\boldsymbol{d_r}$ und $\boldsymbol{d_a}$ berücksichtigt, so wären die Gleichungen 28 und 29 zu erweitern:

$$c_e = a_g \cdot t_e \cdot s + d_r \cdot t_r \qquad \text{Gl.(30)}$$

$$\text{und } c_z = \frac{n \cdot t_z}{t_r} + d_a \cdot a_g. \qquad \text{Gl.(31)}$$

Der Aufwand ist für beide Verfahren gleich, wenn

$$a_g \cdot t_e \cdot s + d_r \cdot t_r = \frac{n \cdot t_z}{t_r} + d_a \cdot a_g \text{ gilt.} \qquad \text{Gl.(32)}$$

Werden die Terme $d_r \cdot t_r$ und $d_a \cdot a_g$ vernachlässigt und für die aufwendigere Berechnung der Neuronenmodelle in einer ereignisgetriebenen Simulation mit Faktor $\boldsymbol{j}$ bezüglich des Rechenzeitaufwands in einer Zeitscheibensimulation angegeben, so ergibt sich der Zusammenhang

$$a_g \cdot j \cdot t_z \cdot s = \frac{n \cdot t_z}{t_r} \Rightarrow a_g \cdot j \cdot s = \frac{n}{t_r} \Rightarrow a_g = \frac{n}{t_r \cdot j \cdot s}. \qquad \text{Gl.(33)}$$

Die Aktivität im Gesamtsystem muss also dem Quotienten aus der Neuronenzahl, dividiert durch das Produkt von Zeitauflösung, durchschnittliche Zahl der Nachfolger und dem erhöhten Rechenzeitbedarf eines Modells in einer ereignisgetriebenen Simulation entsprechen, damit der Aufwand für beide Verfahren gleich ist.

In Abhängigkeit von den Modellen spielt der Faktor j eine untergeordnete Rolle und liegt damit nahe eins. Dem nun stark vereinfachten Zusammenhang

$$a = \frac{n}{t_r \cdot s} \qquad \text{Gl.(34)}$$

ist zu entnehmen, dass bei einem Netz mit geringer Aktivität entweder eine geringe Neuronenzahl von Neuronen, eine große durchschnittliche Zahl an Verbindungen zu nachfolgenden Neuronen vorliegen oder eine sehr grobe Zeitauflösung verwendet werden muss, um annähernd gleiche Rechenzeiten für eine Zeitscheibensimulation und eine ereignisgetriebene Simulation zu erreichen. Treffen auf ein Netzwerk sowohl die Voraussetzungen einer geringen Aktivität als auch einer spärlichen Vernetzung zu, so ist für Netzwerke nennenswerter Größe, also einer größeren Anzahl von Neuronen, lediglich durch eine starke Reduktion der zeitlichen Auflösung der Aufwand der Zeitscheibensimulation in die Größenordung einer ereignisgetriebenen Simulation zu bringen.

3.2.2. Zeitliche Auflösung und numerische Genauigkeit

Auch wenn die zeitliche Auflösung bei einer ereignisgetriebenen Simulation nahezu beliebig erhöht werden kann, so sind einige Punkte in diesem Zusammenhang zu berücksichtigen:

Einerseits steigt die Laufzeit der einzelnen Berechnungen für zukünftige Ereignisse an, da die Zeitpunkte, zu denen ein Neuron aktiv wird, genauer berechnet werden müssen, andererseits ist die zeitliche Auflösung durch die numerische Repräsentation der Zeitwerte beschränkt. Daher bedingen höhere zeitliche Auflösungen auch Repräsentationen mit mehr Bitstellen als geringere Auflösungen.

Die gewählte numerische Repräsentation beeinflusst zudem die relative Genauigkeit auf unterschiedliche Weise. Die Zeitwerte können in Standardrechnersystemen entweder durch einen Fließkommawert oder durch einen Festkommawert modelliert werden. Der Fließkommawert hat den Vorteil, dass aufgrund des größeren Wertebereichs längere Simulationen möglich sind, jedoch auch den Nachteil, dass mit fortschreitender Simulationszeit die Genauigkeit der Repräsentation abnimmt. Festkommawerte bieten den Vorteil, dass mit einer festen Auflösung über den gesamten Wertebereich gearbeitet werden kann, die maximale Simulationsdauer fällt in Abhängigkeit der Auflösung jedoch geringer aus. Fordert man jedoch eine minimale Zahl Nachkommastellen auch für die Repräsentation mit Fließkommawerten, so beschränkt dies auch die maximale Simulationsdauer entsprechend.

Darüber hinaus stellt sich bei beiden Darstellungen mit zunehmender Simulationsdauer eine kumulierte Ungenauigkeit ein, da alle Zeitpunkte mit einer beschränkten Genauigkeit, respektive Ungenauigkeit berechnet werden und alle folgenden Ereignisse auf Basis der bereits ungenau berechneten Ereignisse bestimmt werden. Der kumulative Fehler hängt dabei von der Aktivität im Netzwerk ab, da er lediglich bei einer Berechnung eines neuen Ereignisses auftritt, er ist im Gegensatz zur zunehmenden Ungenauigkeit der Fließkommadarstellung unabhängig vom absoluten Simulationsfortschritt. Neben den dargestellten Repräsentationen sind natürlich noch komplexere Datentypen denkbar, die durch Nachführen eines Zeitfensters, welches sich an den im System größten Zeitabständen orientiert, sowohl eine hohe Genauigkeit bieten als auch die Abhängigkeit von der absoluten Simulationszeit vermeiden. Da Zeitwerte innerhalb der Simulation sehr oft verarbeitet werden, kann ein komplexerer Datentyp jedoch zu einer deutlichen Reduktion der Simulationsgeschwindigkeit führen. Zudem ist es schwierig, das notwendige Zeitfenster zu bestimmen, da nicht zuletzt auch die logischen Prozesse Informationen über ihre maximale Verzögerung bereitstellen müssen.

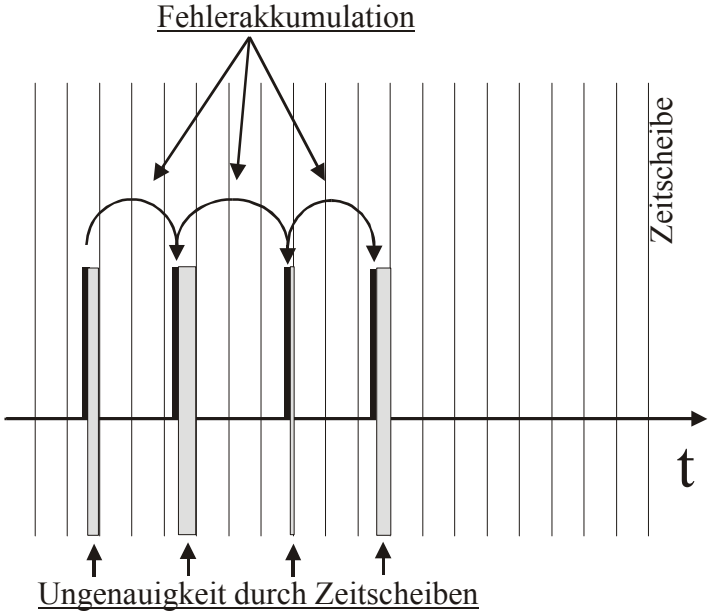


Abbildung 24: Quellen numerischer und verfahrensbedingter Abweichungen

In eine Zeitscheibensimulation wird grundsätzlich nur mit der durch den Zeitscheibenabstand gegebenen Genauigkeit gerechnet, dadurch werden zum Beispiel die Zeitpunkte, zu denen die Aktionspotentiale ausgelöst werden, ungenau abgebildet. Dies führt einerseits zu einer absoluten Ungenauigkeit des einzelnen Feuerzeitpunkts und andererseits zu einem kumulativen Fehler, da die Berechnungen folgender Feuerzeitpunkte auf den fehlerbehafteten Feuerzeitpunkten früherer Berechnungen basieren. Der kumulative Fehler hängt in diesem Fall jedoch nicht von der Aktivität im simulierten Netzwerk ab, sondern nur von der gewählten Zeitauflösung für die Zeitscheiben. Zudem nimmt bei der Verwendung von Fließkommawerten auch bei diesem Verfahren, genauso wie bei der ereignisgetriebenen Simulation, die Genauigkeit mit fortschreitender Simulationszeit ab.

Der kumulative Fehler kann in der ereignisgetriebenen Simulation grundsätzlich nicht umgangen werden, da der Aufruf der logischen Prozesse bedarfs- und zeitgesteuert über die Ereignisse aus der Ereignisliste erfolgt. Nur so kann die geringe Aktivität eines Systems ausgenutzt werden. Lediglich Simulationsverfahren, welche die Berechnung der logischen Prozesse nicht über zeitlich terminierte Ereignisse steuern, sondern bei denen die Berechnungen auf Basis der inneren Zustände der einzelnen Prozesse erfolgen, können den kumulierten Fehler umgehen. Bei solchen Simulationsverfahren handelt es sich um eine Zwischenform der beiden vorgestellten Simulationsverfahren, denn um die Zustandsänderung in allen Prozessen zu berücksichtigen, muss in jeder Iteration der Simulation jeder Prozess aufgerufen und berechnet werden. Es gibt jedoch keine Zeitscheiben, zu denen diese Aufrufe erfolgen. Vielmehr bewegt sich die Simulation von einer wesentlichen Zustandsänderung des Gesamtsystems zur nächsten. Dies kann zum Beispiel das Feuern eines Neurons sein bzw. der Wegfall eines postsynaptischen Potentials, welches durch das Feuern ausgelöst wurde. Es liegt jedoch auch auf der Hand, dass das ein solches Simulationsverfahren sehr eng an das zu simulierende Neuronenmodell gekoppelt ist, da innere Zustandsänderungen des Neuronenmodells Ereignisse in der Simulation repräsentieren. Das Verfahren außerdem nicht für beliebige Neuronenmodelle einsetzbar, da Zeitpunkte identifizierbar sein müssen, zu denen der Zustand des Neurons auf einen definierten Wert zurückgesetzt wird.

Die Bedingung wird zum Beispiel von Neuronenmodellen erfüllt, bei denen durch die Auslösung eines Aktionspotentials das innere Potential zurückgesetzt wird und nach der absoluten Refraktärphase immer wieder von dem gleichen Ausgangswert akkumuliert wird. Weist ein Prozess ein solches Verhalten auf, so können die Berechnungen jeweils auf die Zeitpunkte bezogen werden, an denen das innere Potential zurückgesetzt wird. Diese Zeitpunkte fungieren gewissermaßen als neue Nullpunkte auf der Zeitachse. Da die daraus resultierende, relative Zeitachse einen sehr eingeschränkten Wertebereich hat, kann in diesem Fall mit einer Fließkommadarstellung eine hohe Genauigkeit erreicht werden.

Das beschriebene Verfahren sei an einem Neuronenmodell verdeutlicht, welches Rechteckpulse definierter Länge aussendet, wenn ein interner fester Schwellenwert überschritten wird. Das innere Summenpotential dieses Modells wird durch die Summation der einlaufenden Rechteckpulse der vorgeschalteten Neuronen bestimmt und auf Null zurückgesetzt, sobald der Schwellenwert erreicht ist. Die Summation findet nur statt, wenn das Neuron nicht selber einen Puls aussendet. Das gesamte Netzwerk lässt sich unter den gegebenen Randbedingungen in sendende und empfangende Neuronen unterteilen. Die Zerlegung der Neuronen in sendende und empfangende Neuronen wechselt mit dem Fortschritt der Simulation. Es reicht aus, erneute Berechnungen durchzuführen, sobald sich diese Zerlegung verändert, das heißt wenn entweder eines der Neuronen be-

ginnt, einen Puls auszusenden, oder der Puls eines Neurons endet. Um die Berechnungen zwischen diesen Punkten durchzuführen kann der Startzeitwert immer auf Null gesetzt und der nächste Zeitpunkt, zu dem sich die Zerlegung ändert, unter Verwendung einer Fließkommadarstellung mit hoher Genauigkeit berechnet werden. Dabei wird die Berechnung nicht auf absolute Zeitwerte bezogen und das Ergebnis ist immer der relative Abstand von der zuletzt gültigen Zerlegung zur nächsten. Werden die so ermittelten Abstände zur Dokumentation der Simulation summiert, besteht für die Repräsentation dieser Zeitwerte aber nach wie vor das Problem wachsender Ungenauigkeit, bei wachsender Dauer der Simulation. Allein der kumulative Fehler ist somit weitgehend minimiert.

Die maximal mögliche Dauer einer Simulation hängt einerseits von der gewählten Auflösung und andererseits von der Genauigkeit der gewählten Zahlendarstellung ab. Für vorzeichenlose Festkommawerte ist dieser Zusammenhang gegeben durch $2^{Bitstellen} \cdot Zeitauflösung(\text{sec})$. Für eine Fließkommadarstellung können nach einer IEEE-Norm normalisierte und denormalisierte Repräsentationen unterschieden werden. Beim denormalisierten Fall stehen mehr Nachkommastellen zur Verfügung. In der Praxis kann jedoch der Einfachheit halber von einer normalisierten Repräsentation ausgegangen werden, die für 32bit Fließkommazahlen eine Genauigkeit von 6 Dezimalstellen und für 64bit von 15 Dezimalstellen aufweist. Die folgenden Tabellen geben für verschiedene Zeitauflösungen die benötigte Zahl Nachkommastellen, bezogen auf ein Sekunde, und die maximale Simulationsdauer für verschiedene Wortbreiten an. Der Aufstellung für Fließkommazahlen liegt eine normalisierte Repräsentation zugrunde.

Auflösung	Nachkommastellen	Festkomma 32Bit	Festkomma 64Bit
1ps	12	4,3 ms	213 Tage
1ns	9	4,3 s	584 Jahre
1µs	6	1 h 11 Min.	...
50µs[22]	5	2 t 11 h 39 Min.	...
1ms	3	49 Tage 17 h	...

Tabelle 2: Simulationszeiten für verschiedene Festkommadarstellungen und Zeitauflösungen

Auflösung	Nachkommastellen	Fließkomma 32Bit	Fließkomma 64Bit
1ps	12	--	16 Min. 40 sec
1ns	9	--	11 Tage 13 h
1µs	6	1 s	31 Jahre 259 Tage
50µs	5	10 s	317 Jahre
1ms	3	16 Min. 40 s	...

Tabelle 3: Simulationszeiten für verschiedene, normalisierte Fließkommadarstellungen und Zeitauflösungen

[22] In vielen GENESIS-Simulationen verwendete Zeitauflösung.

Für Rechnersysteme, die eine 64-Bit-Architektur besitzen, liegen die möglichen Simulationszeiten selbst für eine Auflösung von 1ps weit jenseits üblicher Simulationslaufzeiten. Mit 32-Bit Systemen sind Auflösungen bis hin zu 1µs sinnvoll. Wird die Standardauflösung des Simulators GENESIS (50 µs) verwendet, so könnte mindestens 2 Tage simuliert werden.

Wird eine ereignisgetriebene Simulation mit der vorgestellten Modellierung der Ereignisse und Prozesse zur Simulation pulsverarbeitender neuronaler Netze eingesetzt, so kann von der spärlichen Vernetzung und der geringen Aktivität dieser Netze profitiert werden, da die zu investierende Rechenzeit im Wesentlichen von der Aktivität im zu simulierenden Netzwerk abhängt, nicht jedoch von der Zahl der darin enthaltenen Neuronen. Zudem kann mit nahezu beliebiger zeitlicher Genauigkeit simuliert werden, da diese von der Genauigkeit der berechneten Zeitstempel abhängt, nicht jedoch von einem festen Zeitintervall, wie dies in einer zeitgetrieben Simulation der Fall ist.

3.2.3. Verarbeitung kontinuierlicher Signale

Sich kontinuierlich verändernde Signale, auch analoge Signale genannt, werden bei einer Zeitscheibensimulation mit einer Rate abgetastet, die dem Kehrwert des äquidistanten Abstands der Zeitscheiben entspricht. Es bedarf keiner weiteren Vorkehrungen für die Behandlung solcher Signale. Die Genauigkeit, mit der die Signale im System aufgelöst werden, ist durch die Abstände der Zeitscheiben festgelegt.

In einer ereignisgetriebenen Simulation dagegen müssen kontinuierliche Signale besonders behandelt werden, da nicht in allen Fällen eindeutig festgestellt werden kann, wann die Veränderung eines solchen Signals einen wesentlichen Einfluss auf den Zustand des Systems hat. Im ungünstigsten Fall muss das kontinuierliche Signal ähnlich wie in der Zeitscheibensimulation mit einer festen Rate abgetastet werden, wobei jeder Abtastpunkt zu einem Ereignis führt. Es liegt auf der Hand, dass in einem solchen Falle die ereignisgetriebene Simulation keine Vorteile bietet bzw. sogar zu einer deutlichen Verlängerung der Simulationszeit führen kann.

Andererseits besteht die Möglichkeit, das kontinuierliche Signal in einen eigenen logischen Prozess zu kapseln und innerhalb des Prozesses eine Abtastung vorzunehmen, um durch diesen Prozess Ereignisse in das Simulationssystem einzuspeisen, die sich aus dem Verlauf des Signals ergeben. Insbesondere, wenn der Verlauf des Signals im Voraus bekannt ist und z. B. nicht durch einen Sensor oder eine ähnliche Echtzeitquelle geliefert wird, kann der logische Prozess eine Vorausberechnung der daraus resultierenden Ereignisse vornehmen.

Aufgrund der großen Zahl von Ereignissen, die entstehen, wenn Abtastwerte in der ereignisgetriebenen Simulation verarbeitet werden, sollten kontinuierliche Signale immer in logischen Prozessen gekapselt werden.

3.2.4. Stochastische Ereignisse – spontane Aktivität

Stochastische Prozesse, wie sie zum Beispiel durch spontane Aktivitäten von Neuronen entstehen können, müssen genauso wie kontinuierliche Signale bei einer ereignisgetriebenen Simulation gesondert behandelt werden. Für eine Zeitscheibensimulation fällt die Behandlung solcher Ereignisse wieder einfach aus, da sie nur im Rahmen der gegebenen Zeitauflösung entstehen und innerhalb einer Zeitscheibe verarbeitet werden.

In einer ereignisgetrieben Simulation bietet es sich an, die Abschnitte des Systems, in denen stochastische Ereignisse entstehen können, in einen oder mehreren logischen Prozessen zu kapseln. Zudem müssen diese Prozesse zu Beginn der Simulation mindestens einmal aufgerufen werden, damit sie einerseits das nächste von ihnen erzeugte Ereignis und andererseits ein zur gleichen Zeit geplantes Ereignis zum erneuten Aufruf in die Ereignisliste einfügen. Entstehen die Ereignisse nicht im Rahmen eines berechenbaren Zufallsprozesses, sondern werden sie extern zugeführt, so ist die ereignisgetriebene Simulation zeitlich an diese Signale gebunden. Die Simulation kann nur bis zum letzten externen Ereignis fortschreiten, es sei denn, es wird ein *look ahead* angegeben, bis zu dem mit keinen weiteren Ereignissen zu rechnen ist. Kann die Simulation hingegen den zugeführten Ereignissen nicht schnell genug folgen, so sind diese zwischenzuspeichern.

3.2.5. Behandlung von Verzögerungen im Netzwerk

Die Signalfortleitung über die Verbindungen im Cortex laufen stets mehr oder weniger verzögert ab. Verursacht wird dies einerseits durch axonale Verzögerungen, welche abhängig von der Dicke des Axons und seiner Myelinisierung sind, und andererseits durch synaptische Verzögerungen, wenn beispielsweise bei den überwiegend vorhandenen chemischen Synapsen Transmitterstoffe gebildet und übertragen werden müssen (→ Abschnitt 2.1.4).

Im Rahmen der ereignisorientierten Simulation lassen sich bei einer senderorientierten Verteilung der Ereignisse die Verzögerungen im Netzwerk sehr einfach behandeln, da auf den Zeitstempel des ausgehenden Ereignisses eines Neurons eine beliebige Verzögerung addiert werden kann, bevor die Ereignisse in die Ereignisliste gelangen. Bei diesem Verfahren wird die axonale und die synaptische Verzögerung zu der charakteristischen Verzögerung zwischen einem Neuron und einer bestimmten Synapse eines Nachfolgeneurons aufsummiert. Natürlich lassen sich axonale und synaptische Verzögerungen auch getrennt behandeln. Für die ereignisgetriebene Simulation interessiert jedoch lediglich die Summe der Verzögerungen. Eine Ereignisliste – als zentraler Dreh- und Angelpunkt der Simulation – verwaltet grundsätzlich verzögerte Ereignisse.

Bei einer Zeitscheibensimulation verursacht die Behandlung solcher Verzögerungen einen zusätzlichen Aufwand, da Veränderungen bzw. Ereignisse, die innerhalb einer Zeitscheibe berechnet werden, nicht unmittelbar als Ausgangspunkt der Berechnungen in der nächsten Zeitscheibe dienen, sondern erst nach einer bestimmten Verzögerung, also einige Zeitscheiben später, wieder bei den Berechnungen berücksichtigt werden sollen. Dieses Problem kann durch verschiedene Implementierungsansätze gelöst werden. Im Allgemeinen sind dies jedoch Varianten lokaler Ereignislisten, die jedes Neuron besitzt. Die verzögerten Ereignisse werden dort eingetragen und in jedem Zeitschritt überprüft, ob das erste, und damit früheste Ereignis bei der Berechnung der aktuellen Zeitscheibe berücksichtigt werden muss. Letztlich führt dies dazu, dass für die Zeitscheibensimulation neben dem ohnehin vorhandenen Rechenzeitaufwand ein zur ereignisgetriebenen Simulation vergleichbarer Mehraufwand entsteht, da ebenso Ereignislisten verwaltet werden müssen. Darüber hinaus lassen sich die Verzögerungen nur als Vielfaches der Zeitscheiben ausdrücken, wohingegen innerhalb der ereignisgetriebenen Simulation im Rahmen der gewählten Zeitauflösung beliebige Verzögerungen verwendet werden können.

3.3. Optimierung der ereignisgetriebenen Simulation

3.3.1. Zentrale Simulationsschleife

In der einfachsten Form der ereignisgetriebenen Simulation wird durch die Simulationsverwaltung jeweils das frühste Ereignis aus der Ereignisliste entnommen und dem betreffenden logischen Prozess zur Verarbeitung zugeführt. Der logische Prozess speist dann seinerseits Ereignisse in die Ereignisliste ein, die aufgrund der Verarbeitung des eingegangenen Ereignisses entstanden sind. Erst wenn dieser Prozess abgeschlossen ist, übernimmt die Verwaltung wieder die Kontrolle des Ablaufs und entnimmt wiederum das früheste Ereignis aus der Liste.

Je kürzer die Verarbeitungszeit für ein Ereignis ist, desto schwerer wiegt der zusätzliche Aufwand, der für die Verwaltung aufgebracht werden muss. Nicht nur aus diesem Grunde liegt es nahe, den Verwaltungsaufwand so weit wie möglich zu reduzieren. Hierfür bietet sich die gemeinsame Verarbeitung mehrerer Ereignisse in einem Block an, da die Verwaltung dann nicht mehr für jedes einzelne Ereignis aufgerufen zu werden braucht.

Ohne Schwierigkeiten lässt sich eine solche Blockverarbeitung mit gleichzeitigen Ereignissen umsetzen, da in einem solchen Falle keine weiteren Randbedingungen mehr berücksichtigt werden müssen. Daher beinhaltet bereits die Grundbetriebsart des später vorgestellten Simulationssystems auch die Verarbeitung mehrerer gleichzeitiger Ereignisse. Findet eine Simulation von Modellen statt, die sich möglichst getreu an biologischen neuronalen Netzen orientieren, so ist die Wahrscheinlichkeit gleichzeitiger Ereignisse nahe Null und der Gewinn dieser Maßnahme vernachlässigbar, nicht jedoch für stärker vereinfachte Modelle, in denen z. B. viele gleiche Verzögerungswerte auftreten.

Ein weitaus größeres Optimierungspotential kann ausgeschöpft werden, wenn es gelingt eine Blockverarbeitung mehrerer, nicht gleichzeitiger Ereignisse zu implementieren. Dies begründet sich nicht nur aus der Tatsache, dass mit einer solchen Blockverarbeitung auch im Falle biologienaher Modelle eine Beschleunigung zu erreichen ist, sondern können hierdurch auch sonst notwendige Kontrollereignisse reduziert oder ganz vermieden werden.

Bei solchen Kontrollereignissen handelt es sich im Wesentlichen um so genannte *wake-up* Ereignisse, die dazu verwendet werden, bereits berechnete Pulse, die jedoch nicht sicher sind, in der Ereignisliste zu vermerken. Der Puls wird erst dann in die Ereignisliste eingetragen, wenn der berechnete Zeitpunkt erreicht ist und der Puls nicht aufgrund eines weiteren Eingangsereignisses in der Zwischenzeit revidiert wurde. Sicher sind hingegen Ereignisse, welche innerhalb eines Horizonts liegen, der durch den minimalen *look-ahead* über alle Eingänge eines logischen Prozesses definiert wird.

In Abbildung 25 ist eine typische Verarbeitungssituation dargestellt, in der ein *wake-up* Ereignis erzeugt wird. Auf ein Neuron treffen kurz hintereinander zwei Pulse. Der zweite Puls trifft zwischen dem zuletzt eingegangenen Puls und dem daraus berechneten Feuerzeitpunkt ein. Da der zweite Puls, wie der erste Puls, einen positiven postsynaptischen Potentialbeitrag auslöst, wird der Schwellenwert früher erreicht als mit dem ersten Puls alleine. Der zuvor berechnete Feuerzeitpunkt wird damit ungültig. Auf die gleiche Weise kann ein negatives postsynaptisches Potential die Aussendung eines Pulses gänzlich unterbinden.

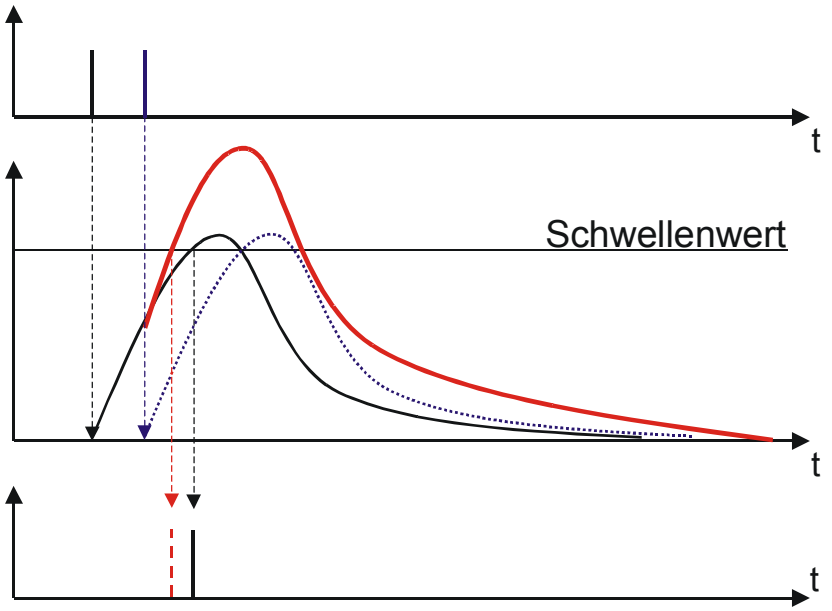


Abbildung 25: Verschiebung des Feuerzeitpunkts durch ein erneut eintreffendes Ereignis.

Daher ist einem solchen Falle ein *wake-up* Ereignis für den zuerst berechneten Feuerzeitpunkt in der Ereignisliste einzutragen. Gelänge es jedoch, die beiden kurz hintereinander eintreffenden Ereignisse zu bündeln, so müsste kein *wake-up* Ereignis generiert werden und der endgültige Feuerzeitpunkt könnte vom logischen Prozess direkt zurückgeliefert werden.

Ohne die Konsistenz der Simulation zu verletzen, lässt sich jedoch nur eine begrenzte Anzahl Ereignisse zusammenfassen, da sichergestellt sein muss, dass keine Wechselwirkungen zwischen den Verarbeitungen der einzelnen Ereignisse bestehen. Im einfachsten Falle ist diese Bedingung erfüllt, wenn strukturell keine Abhängigkeit zwischen zwei logischen Prozessen existiert, also keine direkte oder indirekte Verbindung existiert. Liegen hingegen direkte oder indirekte Verbindungen zwischen den Prozessen vor, so muss für jedes Ereignis, welches der Ereignisliste entnommen wird, geprüft werden, ob die daraus zu berechnenden Ereignisse einen Einfluss auf die Berechnung bereits entnommener Ereignisse ausüben könnten.

Zur Verdeutlichung sei eine Anordnung von logischen Prozessen wie in Abbildung 26 angenommen. Ein Ereignis zum Zeitpunkt $t_1^{ein} = t_0$, für den Prozess 1, sei das früheste Ereignis aus der Ereignisliste. Dieses Ereignis kann immer sicher verarbeitet werden und führt potentiell zu einem Ausgangsereignis zum Zeitpunkt t_1^f . Das Ausgangsereignis kann frühestens zur gleichen Zeit wie das eingehende Ereignis oder zu einem Zeitpunkt entstehen, der um die Reaktionszeit t_1^{reak} verzögert ist. Bleibt das Ereignis aus, so entspricht dies einer unendlich langen Reaktionszeit. Einen nachfolgenden Prozess j würde das Ausgangsereignis nach einer Verzögerung Δ_{ij} erreichen (Im Beispiel würde das Ausgangsereignis Prozess 4 nach der Verzögerung Δ_{14} erreichen).

Es gilt also

$$t_i^f = t_i^{ein} + t_i^{reak} \text{ mit } 0 \le t_i^{reak} \le \infty$$

und $t_j^{ein} = t_i^f + \Delta_{ij}$.

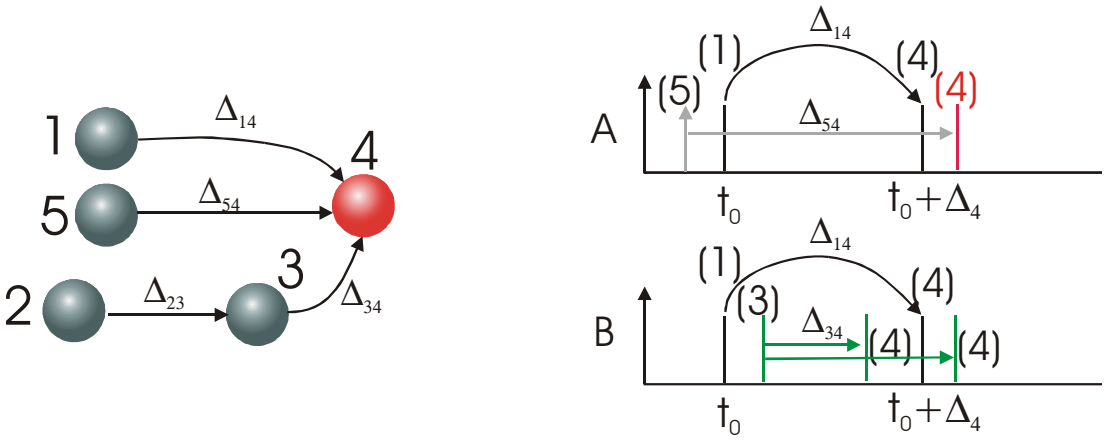


Abbildung 26: Direkte und indirekte Verbindungen von Prozessen

Die Reaktionszeit ist eine dynamische Größe, da sie in der Regel vom inneren Zustand des Prozesses und der Gewichtung des Eingangspulses abhängt. Deshalb lässt sich über t_i^f keine sichere Aussage treffen, solange das Ereignis noch nicht verarbeitet wurde. Lediglich die Extremfälle mit $t_i^{reak} = 0$ und $t_i^{reak} = \infty$ sind bekannt, da für $t_i^{reak} = 0$ $t_i^f = t_i^{ein}$ gilt und für $t_i^{reak} = \infty$ kein Ausgangspuls entsteht. Damit ist bekannt, dass bei einem Nachfolger j frühestens ein Ereignis zum Zeitpunkt $t_j^{ein} = t_i^{ein} + \Delta_{ij}$ eintrifft. Im Beispiel gilt daher $t_4^{ein} = t_1^{ein} + \Delta_{14}$.

Bei der Entnahme eines weiteren Ereignisses aus der Ereignisliste muss nun geprüft werden, ob dieses Ereignis von dem Prozess i = 1 zu verarbeiten ist, der auch das zuvor entnommene Ereignis zu verarbeiten hat, oder ob es von dem Prozess j = 4 zu verarbeiten ist, welcher Ziel des potentiellen Eingangsereignis zum Zeitpunkt t_4^{ein} ist. Handelt es sich um den gleichen Prozess (i = 1), so kann das Ereignis in diesem Fall gemeinsam mit dem zuvor entnommenen Ereignis verarbeitet werden – sofern keine Rückkopplungsverbindung existiert. Ist jedoch der Prozess j = 4 das Ziel oder existiert eine Rückkopplung, so muss zudem geprüft werden, ob das aktuell entnommene Ereignis vor oder nach dem durch das erste Ereignis bestimmten Zeitpunkt t_4^{ein} stattfindet. Liegt es davor, so kann es gemeinsam mit dem zuvor entnommenen Ereignis verarbeitet werden. Liegt es zeitlich später, so kann die Verarbeitung nicht gemeinsam mit dem ersten Ereignis stattfinden, da sich der Zustand des Prozesses j = 4 aufgrund des potentiellen Ereignisses zum Zeitpunkt t_4^{ein} ändern könnte und damit eine parallele Ausführung der Berechnungen zu falschen Ergebnissen führen kann. Diese Situation liegt zum Beispiel im Fall A vor, bei dem aus dem letzten Verarbeitungszyklus von Prozess 5 ein Ereignis für den Prozess 4 in der Ereignisliste abgelegt wurde, welches nach dem potentiellen Eingangsereignis stattfinden würde, das durch den Prozess 1 erzeugt werden kann.

Das Ereignis, welches von Prozess 3 im Fall B verarbeitet wird, kann hingegen in jedem Fall gemeinsam mit dem Ereignis für Prozess 1 verarbeitet werden. Hierbei ist jedoch zu beachten, dass durch die Verarbeitung des Ereignisses für Prozess 3 das potentiell früheste Eingangsereignis für Prozess 4 sich zu früheren Zeitpunkten verschieben kann. Dies ist der Fall, wenn $t_3^{ein} + \Delta_{34} < t_1^{ein} + \Delta_{14}$ gilt. Mit $t_j^{ein} = t_i^f + \Delta_{ij}$ kann für jeden nachfolgenden Prozess j das potentiell früheste Eingangsereignis bestimmt werden das

durch die Verarbeitung des aktuellen Ereignisses entstehen würde. Das Minimum aus allen t_j^{ein} für einen Prozess j definiert schließlich den kritischen Zeitpunkt bis zu dem weitere Ereignisse für diesen Prozess der Ereignisliste, zwecks einer gemeinsamen Verarbeitung, entnommen werden können. D.h. für die Menge der entnommenen Ereignisse die von den Prozessen i verarbeitet werden müssen, können die kritischen Zeitpunkte für jeden nachfolgenden Prozess j zu

$$t_j^{krit} = Min(t_i^{ein} + \Delta_{ij})$$

bestimmt werden. Hierdurch werden jedoch nur direkte Abhängigkeiten erfasst. Indirekte Abhängigkeiten, wie sie sich z. B. durch die Verbindung von Prozess 2 und 3 ergeben, können ebenso zu einer Verschiebung des kritischen Zeitpunkts für den Prozess 4 führen, auch wenn für Prozess 3 kein Ereignis vorliegt.

Da jedes neu zu entnehmende Ereignis in beschriebener Weise geprüft werden müsste, wächst der Aufwand für eine vollständige Untersuchung mit zunehmender Zahl entnommener und gemeinsam zu verarbeitenden Ereignisse. Der Aufwand ist dabei insbesondere durch die Überprüfung der eingangs- bzw. ausgangsseitigen Abhängigkeiten zwischen den Prozessen bestimmt, weil für jeden Prozess j das Minimum aus allen t_j^{ein} bestimmt und verwaltet werden muss.

Wird darauf verzichtet die Abhängigkeiten zwischen den Prozessen zu untersuchen, so muss nur noch ein Minimum für alle Prozesse verwaltet werden, da grundsätzlich davon ausgegangen werden muss dass eine Abhängigkeit zwischen den Zielprozessen der entnommenen Ereignisse besteht. Das heißt, dass für jedes entnommene Ereignis

$$t^{krit} = t_i^{ein} + \Delta_i^{\min}$$

bestimmt werden muss, wobei $\Delta_i^{\min}$ das Minimum aus allen Verzögerung zu den Nachfolgern des aktuell betrachteten Prozesses i ist. Dieser Wert kann während der Initialisierung der Simulation bestimmt und gespeichert werden.

Bei diesem Verfahren wird also ein kritischer Zeitpunkt bestimmt, bis zu dem Ereignisse zwecks einer gemeinsamen Verarbeitung der Liste entnommen werden können. Dieser kritische Zeitpunkt wird durch die Entnahme eines Ereignisses unter Umständen zu früheren Zeitpunkten verschoben. Sobald ein zu entnehmendes Ereignis einen Zeitstempel aufweist, der größer oder gleich dem aktuellen kritischen Zeitpunkt ist, wird die Entnahme gemeinsam zu verarbeitender Ereignisse beendet und die bis dahin gesammelte Menge Ereignisse gemeinsam verarbeitet.

Grundsätzlich ließe sich dieses Verfahren noch erweitern indem für jedes entnommene Ereignis beim dem dieses Ereignis verarbeitenden Prozess angefragt wird, wann frühestens mit einer Aktivität des Prozesses zu rechnen ist, so dass bei der Bestimmung der kritischen Zeitpunkte nicht immer von dem ungünstigsten Fall, nämlichen der instantanen Aktivität des Prozesses, also $t_i^{reak} = 0$, ausgegangen werden muss. Hierbei könnte auch ausgenutzt werden, dass der Prozess u.U. ohne aufwendige Berechnungen feststellen kann, dass er aufgrund des Ereignisses nicht aktiv werden wird. Abhängig von der Komplexität des Modells ist der Aufwand für einen solchen Aufruf ähnlich hoch wie die Berechnung des Modells selber. Auf Grund der kurzen Berechnungszeiten der in dieser Arbeit betrachteten Modelle wurde daher auf eine solche Erweiterung des Verfahrens verzichtet.

3.3.2. Neuronenmodelle

Anders als bei der Zeitscheibensimulation, bei der zu jeder Zeitscheibe der Zustand des zu simulierenden Neuronenmodells aktualisiert und dann getestet wird, ob der Schwellenwert überschritten ist, wird der Zustand der Neuronenmodelle einer ereignisgetriebenen Simulation in größeren, unregelmäßigen Abständen aktualisiert, da nur eingehende Pulse eine Berechnung des Modells auslösen. Deshalb muss auch eine Vorhersage des zukünftigen Verlaufs der Anregungsfunktion erfolgen, um zu prüfen, ob der Schwellenwert überschritten und somit ein Ausgangspuls erzeugt wird.

Aus den in Abschnitt 2.2 beschriebenen Modellen sei hier auf das Spike-Response-Modell näher eingegangen, um eine optimierte Umsetzung der Algorithmen für eine ereignisgetriebene Simulation zu erläutern. Basierend auf dem in Abschnitt 2.2.4 beschriebenen SRM wurden im Rahmen von [Weber1999] entsprechende Neuronenmodelle für das später vorgestellte Simulationssystem implementiert, welche alle auf dem im Folgenden dargestellten Basisalgorithmus beruhen.

Mit den Gleichungen 9, 11 und 12 (Seite 29) ist die Anregungsfunktion eines SRM gegeben durch

$$h_i(t) = \sum_j w_{ij} \sum_f \varepsilon\left(t - t_j^f\right) + \sum_f \eta_{refr}\left(t - t_i^f\right). \qquad \text{Gl. (35)}$$

Mit τ_i für die Zeitpunkte der eingehenden Pulse, und $\hat{\tau}_i$ für die der ausgehenden Pulse, der Zeitkonstanten $\alpha = \frac{1}{\tau_r}$ und dem einem Ausgangsgewicht V sind die Kernelfunktionen $\varepsilon(t)$ und $\eta(t)$ des SRM durch

$$\varepsilon(t) = \frac{(t-\tau_i)}{\tau_r} \cdot e^{\frac{-(t-\tau_i)}{\tau_r}} = \alpha(t-\tau_i) \cdot e^{-\alpha(t-\tau_i)} \quad \text{Gl. (36)} \qquad \eta_{refr}(t) = Ve^{-\alpha(t-\hat{\tau}_i)} \quad \text{Gl. (37)}$$

gegeben. Die Anregungsfunktion des SRM lässt sich umformen zu

$$\begin{aligned} h_i(t) &= \sum_{\tau_i \le t} W_i \alpha(t-\tau_i)\ e^{-\alpha(t-\tau_i)} - \sum_{\hat{\tau}_i \le t} Ve^{-\alpha(t-\hat{\tau}_i)} \\ &= \left(\sum_{\tau_i \le t} W_i e^{\alpha\tau_i} \alpha t - \sum_{\tau_i \le t} W_i \alpha \tau_i e^{\alpha\tau_i} - \sum_{\hat{\tau}_i \le t} Ve^{\alpha\hat{\tau}_i} \right) e^{-\alpha t} \qquad \text{Gl. (38)} \\ &= \left(A\alpha t + B \right) e^{-\alpha t} \end{aligned}$$

mit den Ersetzungen $A := \sum_{\tau_i \le t} W_i e^{\alpha\tau_i}$ Gl. (39) und $B := -\sum_{\tau_i \le t} W_i \alpha \tau_i e^{\alpha\tau_i} - \sum_{\hat{\tau}_i \le t} Ve^{\alpha\hat{\tau}_i}$ Gl. (40).

Jegliche Zustandsänderung wird durch die Funktionen *A* und *B* bestimmt, deren Berechnungsergebnisse jeweils in gleichnamigen Variablen gespeichert werden. Der Zustand des SRM ändert sich, wenn ein Puls an einem der Eingänge vorliegt oder ein ausgehender Puls generiert wird. Außerdem muss zwischen diesen Ereignissen eine Anpassung der Variablen *A* und *B* erfolgen, welche die Veränderungen durch das Fortschreiten der Zeit von τ_{i-1} auf τ_i erfasst.

Daher kann die Berechnung des SRM in die folgende Teilaufgaben aufgegliedert werden: Verschiebung des Bezugszeitpunkts (*MoveTime*), Verarbeitung eines Eingangspulses (*inSpike*), die Vorhersage des zukünftigen Verlaufs der Anregungsfunktion, um auf

die Überschreitung des Schwellenwerts zu testen (*testAndGet*), und die Generierung eines Ausgangspulses (*outSpikes*). Diese Funktionen formen in der beschriebenen Abfolge den Basisalgorithmus zur Berechnung von Modellen, die sich in der oben beschriebenen Form ausdrücken lassen.

Bei der Verschiebung des Bezugszeitpunkts um *dt* folgt eingesetzt in die Anregungsfunktion für die neue Anregung $h'(t)$

$$\begin{aligned} h'(t) &= h(t+dt) \\ &= \left(A\alpha(t+dt)+B\right)e^{-\alpha(t+dt)} \\ &= \left(Ae^{-\alpha dt}\alpha t + \left(A\alpha dt + B\right)e^{-\alpha dt}\right)e^{-\alpha t} \end{aligned} \qquad \text{Gl. (41)}$$

d.h. für die neuen Zustandsvariablen *A'* und *B'* gilt somit:

$$A' = Ae^{-\alpha dt} \text{ Gl. (42) und } B' = \left(A\alpha dt + B\right)e^{-\alpha dt}. \qquad \text{Gl. (43)}$$

Erfolgt die Verschiebung des Bezugszeitpunkts auf den Zeitpunkt eines eingehenden Pulses, so werden A' und B' nach Gleichung 42 und 43 berechnet und es gilt für den eingehenden Puls $\tau_i = t = 0$. Die Auswirkung des eingehenden Pulses lässt sich an diesem neuen Bezugzeitpunkt durch Hinzufügen eines weiteren Summanden berücksichtigen:

$$A^{neu} = A + W_i \qquad \text{Gl. (44)}$$

$$\text{und } B^{neu} = B \ . \qquad \text{Gl. (45)}$$

Entsprechendes gilt bei der Generierung eines Ausgangspulses, wobei in diesem Falle

$$A^{neu} = A \qquad \text{Gl. (46)}$$

$$\text{und } B^{neu} = B - V \qquad \text{Gl. (47)}$$

gilt, sofern der innere Zustand nach einem ausgehenden Puls nicht zurückgesetzt wird. Ist dies jedoch der Fall, so gilt einfach $A^{neu} = 0$ und $B^{neu} = 0$.

Um potentielle Ausgangspulse vorherzusagen, muss die durch die Anregungsfunktion $h(t)$ bestimmte Kurve diskutiert werden. Dabei können zum Betrachtungszeitpunkt folgende Fälle unterschieden werden:

1. Der aktuelle Wert der Anregungsfunktion $h(0)$ liegt oberhalb des Schwellenwerts. Dies kann der Fall sein, wenn der Zustand des Modells nach einem ausgehenden Puls nicht zurückgesetzt wird. Dann ist unmittelbar nach der absoluten Refraktärphase der Schwellenwert wieder überschritten: $h(0) = \left(A\alpha \cdot 0 + B\right)e^{-\alpha \cdot 0} = B \geq \vartheta$.

2. Für den aktuellen Wert der Anregungsfunktion gilt $h(0) < \vartheta$, das heißt der Schwellenwert kann nur im weiteren Verlauf von $h(t)$ überschritten werden. Gilt in diesem Falle $A = 0$, so fällt die Anregungsfunktion für $B \geq 0$ oder ist negativ, wenn $B < 0$ gilt. Gilt hingegen $A \neq 0$, so ist die Untersuchung der lokalen Extremstellen $\hat{t}$ von $h(t)$ notwendig. Notwendige Bedingung für eine Extremstelle ist, dass die erste Ableitung von $h(t)$ Null ist. Aufgelöst nach $\hat{t}$ folgt:

$$\frac{dh}{dt}(\hat{t}) = A\alpha e^{-\alpha\hat{t}} + \left(A\alpha\hat{t} + B\right)(-\alpha)e^{-\alpha\hat{t}} = 0$$

$$A - \left(A\alpha\hat{t} + B\right) = 0$$

$$\hat{t} = \frac{1 - \frac{B}{A}}{\alpha} \qquad \text{Gl. (48)}$$

Anhand der zweiten Ableitung

$$\frac{d^2h}{d^2t}(\hat{t}) = -\alpha^2 A e^{\frac{B}{A}-1} \neq 0$$

sieht man, dass es sich um eine lokales Extremum handelt. Den so ermittelten Wert von $\hat{t}$ kann man nun in die Anregungsfunktion einsetzen, um den Wert von *h(t)* an der Stelle $\hat{t}$ zu erhalten:

$$h(\hat{t}) = Ae^{\frac{B}{A}-1}.$$

Wäre die Anregungsfunktion an dieser Stelle kleiner Null, so hätte man ein globales Minimum gefunden, da die Anregungsfunktion für $t \to \infty$ gegen Null konvergiert. Für $0 < h(\hat{t}) < \vartheta$ dagegen ist an der Stelle $\hat{t}$ ein globales Maximum. Damit liegen alle Funktionswerte unter dem Schwellenwert. Auch muss $\hat{t} \geq 0$ gelten, da die Funktion bei $\hat{t} < 0$ für $t \geq 0$ gegen Null konvergiert und für $B \geq 0$ die Ungleichung $h(t) \leq h(0) < \vartheta$ bzw. für $B < 0$ die Ungleichung $h(t) < 0 < \vartheta$ gilt. Die letzte Bedingung trifft genau dann zu, wenn $A \geq B$ gilt. Die beiden Bedingungen zusammen sind auch hinreichend für das Auslösen eines Spikes. Durch die Stetigkeit von *h(t)* und mit dem Mittelwertsatz folgt, dass innerhalb des Intervalls $[0,\hat{t}]$ der Schwellenwert erreicht wird. Ist $h(\hat{t}) > \vartheta$, so ist nur der erste Zeitpunkt $\tilde{t}$ wichtig, bei dem gilt $h(\tilde{t}) = \vartheta$:

$$\begin{aligned} \left(A\alpha\tilde{t} + B\right)e^{-\alpha\tilde{t}} &= \vartheta \\ \left(\alpha\tilde{t} + \tfrac{B}{A}\right)e^{-\alpha\tilde{t}} &= \tfrac{\vartheta}{A} \\ \left(\alpha\tilde{t} + \tfrac{B}{A}\right)e^{-\alpha\tilde{t}-\frac{B}{A}} &= \tfrac{\vartheta}{A}e^{-\frac{B}{A}} \end{aligned} \quad .$$

Mit der Abkürzung $x := \alpha\tilde{t} + \frac{B}{A}$ folgt

$$xe^{-x} = \frac{\vartheta}{A}e^{-\frac{B}{A}}$$

Im Folgenden berechnet man die rechte Seite obiger Gleichung und löst mit Hilfe einer Tabelle nach *x* auf, um anschließend

$\tilde{t} = \frac{1}{\alpha}\left(x - \frac{B}{A}\right)$ zu berechnen.

Damit lässt sich die Funktion zur Vorhersage des Feuerzeitpunkts in Pseudocode folgendermaßen darstellen:

```
if B ≥ ϑ then                          { Schwellenwert ist bereits überschritten    }
   t ← 0;                              { zum relativen Zeitpunkt Null               }
   return true;                        { zeigt an, dass Spike vorhergesagt wurde    }
fi;
if A < B or A ≤ 0 then                 { eine der notw. Bedingungen nicht erfüllt   }
```

```
    return false;                    { es wird kein Spike in der Zukunft generiert }
  fi;
  y ← ϑ / A * exp(-B/A);             { Berechne rechte Seite der Gleichung ()      }
  x ← Table(y);                      { Löse Gleichung mit Hilfe einer Tabelle      }
  t ← 1 / α * (x – B/A);             { Berechne Zeitpunkt                          }
  return true;                       { zeigt an, dass Spike vorhergesagt wurde     }
```

3.3.3. Verteilte Simulation

Bei einer verteilten Simulation reicht die Synchronisation über die Ereignisliste des Simulators nicht mehr aus. Um die zeitliche Konsistenz in jedem an der Simulation beteiligten Subsimulator zu gewährleisten, müssen sich die Subsimulatoren untereinander synchronisieren oder zentral über einen Kontroller synchronisiert werden. Eine zentrale Synchronisation verbietet sich jedoch aufgrund des dafür notwendigen Kommunikationsaufwandes. Für eine dezentrale Synchronisation einer ereignisgetriebenen Simulation haben sich die zwei in Abschnitt 3.1.2.2 beschriebenen Verfahren, die konservative und die optimistische Synchronisation, etabliert.

Die Simulation im später vorgestellten Simulationssystem wird konservativ synchronisiert. Bei der konservativen Synchronisation darf ein Subsimulator des verteilten Simulators nur sichere Ereignisse verarbeiten. Ein Ereignis ist dann als sicher zu bezeichnen, wenn zweifelsfrei feststeht, dass von benachbarten Subsimulatoren, die Verbindungen zu dem betrachteten Subsimulator besitzen, keine Ereignisse eintreffen können, die früher als das zu verarbeitende Ereignis stattfinden. Diese Bedingung ist erfüllt, wenn von den benachbarten Subsimulatoren entweder Ereignisse mit einem größeren Zeitstempel vorliegen oder die Subsimulatoren durch die Übermittlung eines *look aheads* versichern, über den Zeitpunkt des betrachteten Ereignisses hinaus keine weiteren Ereignisse mehr zu senden. Der am wenigsten weit fortgeschrittene Nachbar bestimmt letztlich das sichere Fenster, innerhalb dessen Ereignisse verarbeitet werden können. In Abbildung 27 ist dieser Zusammenhang schematisch dargestellt.

Dort wird das sichere Zeitfenster, in dem die Verarbeitung von Ereignissen stattfinden kann, auf dem Rechner vier durch den Rechner eins bestimmt, welcher einen entsprechenden *look ahead* übermittelt hat. Um einerseits die Blockadefreiheit und andererseits eine gute Simulationsperformance zu gewährleisten, wurde in Spikelab eine Mischform zwischen Blockadeerkennung und –vermeidung implementiert. Eine Blockade eines Subsimulators liegt immer dann vor, wenn das Ende seines sicheren Fensters erreicht ist und keine weiteren Nachrichten von den Vorgängern vorliegen, die das sichere Fenster bestimmen. Tritt diese Situation ein, wird ein dreistufiges Verfahren verwendet:

1. Beim Erreichen des sicheren Fensters wird die aktuelle Simulationszeit an alle Nachfolger versendet.
2. Alle Ereignisse, die nicht von der Blockade betroffenen sind werden verarbeitet der maximale *look ahead* für jeden Subsimulator-Ausgang wird mit Hilfe des „Killing“–Algorithmuses bestimmt und versendet.
3. Besteht die Blockade weiter, so wird an den die Blockade verursachenden Vorgänger eine Anfrage bezüglich des Simulationsfortschritts gestellt, auf deren Beantwortung blockierend gewartet wird.

Erreicht ein Subsimulator das Ende des sicheren Fensters und kann nur weitere Ereignisse verarbeiten, wenn entweder ein look ahead oder ein Ereignis von einem benachbarten Subsimulator eintrifft, so versendet der betroffene Subsimulator seine aktuelle Simulationszeit an alle seine Nachfolger. Diese erste Stufe dient dazu, Blockaden aufzulösen, die durch Rückkopplungen (*race coditions*) bedingt sind, und ermöglicht zudem nachfolgenden Subsimulatoren zumindest bis zu dem Zeitpunkt der Blockade ihres Vorgängers plus einem eventuell vorhandenen *look ahead* zu simulieren.

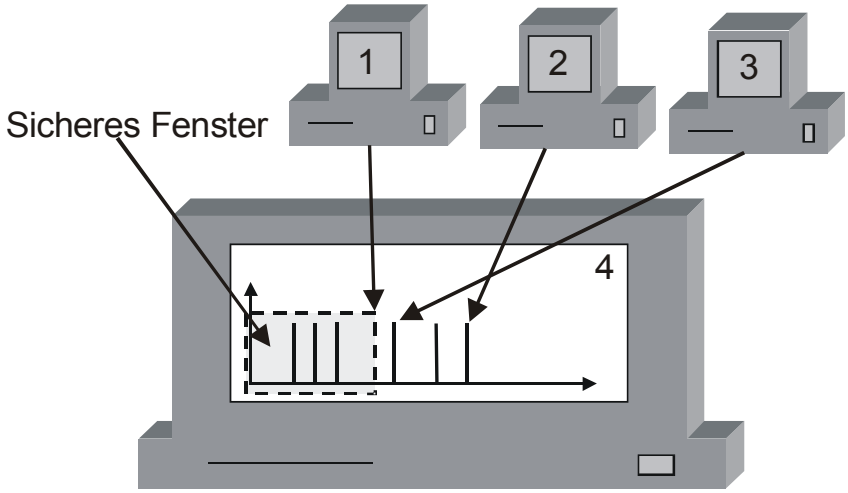


Abbildung 27: Sicheres Fenster für die Verarbeitung von Ereignissen in einer konservativ synchronisierten Simulation

Liegt weiterhin keine Nachricht von dem Vorgänger vor, der das sicherer Fenster der Simulation bestimmt, so werden alle Ereignisse in der Ereignisliste überprüft, ob sie unabhängig von dem Eintreffen externer Ereignisse simuliert werden können. Ein Ereignis ist dann unabhängig von den Eingangsereignissen simulierbar, wenn der kürzeste Pfad zwischen den Eingängen und dem verarbeitenden Neuron eine Summenverzögerung aufweist, die größer ist als der Abstand zwischen dem Zeitstempel des Eingangsereignisses und dem Zeitstempel des zu verarbeitenden Ereignisses. Die kleinstmöglichen Eingangszeitstempel sind aufgrund der vorliegenden *look aheads* oder der zuletzt eingegangenen Eingangsereignisse bestimmt.

Die Bestimmung des kürzesten Pfads kann nun ausgehend von den minimalen Eingangszeitstempeln mit der Verarbeitung der unabhängigen Ereignisse gekoppelt werden. Hierfür wird in einem speziellen Modus des Subsimulators für jeden Eingang einer Partition ein so genanntes *killer event* in der Ereignisliste eingetragen. Das *killer event* besitzt den zuvor bestimmten minimalen Zeitstempel des betreffenden Eingangs. Anschließend wird die serielle Simulationsschleife im Simulator ausgeführt, wobei nun jedoch bei der Verarbeitung eines *killer events* das Zielneuron abgeschaltet wird und an alle Nachfolger des Neurons wiederum ein *killer event* versendet wird. Der Abschaltzeitpunkt bestimmt sich dabei aus dem Zeitstempel des eingehenden *killer events* plus der Verzögerung zum Nachfolgeneuron. Reguläre Ereignisse werden normal verarbeitet, solange das betreffende Neuron noch nicht abgeschaltet ist. Der Algorithmus terminiert, wenn alle Neuronen der Partition abgeschaltet worden sind.

Aus den Abschaltzeitpunkten der Neuronen werden dann die kürzesten Pfade zwischen allen Ein- und Ausgangspaaren bestimmt. Für jedes Ein- und Ausgangspaar wird aus dem minimalen Eingangszeitstempel und der Summenverzögerungen des kürzesten Pfades ein *look ahead* bestimmt. Dieser *look ahead* wird zum Abschluss der zweiten Stufe als Nachricht an alle nachfolgenden Subsimulatoren versendet. Für jedes Ein- und Ausgangspaar des Subsimulators wird nach Abschluss des Simulationslaufs die Summenverzögerung des kürzesten Pfads gespeichert.

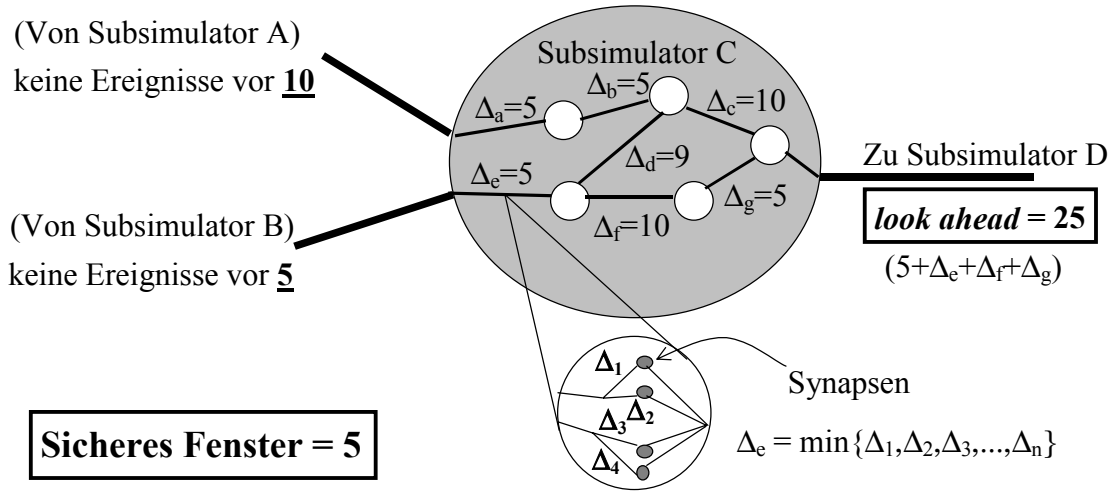


Abbildung 28: Bestimmung des ausgangsseitigen *look aheads* mit Hilfe von Verzögerungen

Abbildung 28 zeigt ein Beispiel für die Bestimmung des maximalen *look aheads* für eine Partition, welche zwei Vorgängerpartitionen (A und B) hat und eine Nachfolgepartition (D). Der Vorgänger A gibt an, bis zum Zeitpunkt 10, und Vorgänger B, bis zum Zeitpunkt 5 keine weiteren Ereignisse zu senden. Damit ist das sichere Fenster für den Subsimulator C auf den Zeitpunkt 5 festgelegt. Wird nun ein Simulationslauf mit *killer events* durchgeführt, so ergibt sich für den Ausgang zum Subsimulator D ein *look ahead* mit dem Zeitstempel 25. Der *look ahead* bildet sich aus der letzten Nachricht vom Vorgänger B mit dem Zeitstempel 5 und der kleinsten Summenverzögerung zwischen diesem Eingang und dem Ausgang zum Nachfolger D. Der kürzeste Pfad wird aus den Verzögerungen Δ_e, Δ_f und Δ_g bzw. Δ_a, Δ_b und Δ_c gebildet. Bei der Bestimmung des *look aheads* wird jedoch in diesem Falle der Pfad Δ_e, Δ_f und Δ_g herangezogen, da von Rechner B bereits zum Zeitpunkt 6 neue Ereignisse erwartet werden können. Für Δ_e ist exemplarisch dargestellt, dass bei mehreren Verbindungen zwischen zwei Neuronen lediglich die kürzeste Verzögerung aller Verbindungen zur Berechnung herangezogen wird, denn die Verwendung von *killer events* stellt sicher, dass ein Neuron immer über den kürzesten Pfad abgeschaltet wird. Dem Nachfolger wird das Minimum aller *look aheads* für Ausgang D – in diesem Falle 25 – übermittelt. Für die Verbindungen A-D und B-D wird jeweils eine Summenverzögerung von 20 gespeichert.

Ein Sonderfall tritt ein, wenn der blockierte Subsimulator beim Eintritt in die zweite Stufe keine weiteren Ereignisse mehr in seiner Ereignisliste hat. In diesem Fall wird die Simulation mit *killer events* nur dann angestoßen, wenn sich entweder seit dem letzten Durchlauf Verzögerungen[23] geändert haben oder bisher noch keine Simulation mit *killer*

[23] Dies ist der Fall, wenn die Verzögerungen während der Simulation durch ein Trainingsverfahren angepasst würden. Dieser Fall wurde in der vorliegenden Arbeit jedoch nicht untersucht.

events erfolgt ist. Denn um einen *look ahead* für jeden Ausgang bestimmen zu können, müssen die Summenverzögerungen zwischen den Ein- und Ausgängen des Subsimulators vorliegen.

Sollte nach der Bearbeitung der ersten beiden Stufen noch immer keine Nachricht des Subsimulators, durch den das sichere Fenster bestimmt wird, eingetroffen sein, so wird an diesen Subsimulator eine Anfrage in Form einer speziellen Nachricht gestellt. Die Simulation im anfragenden Subsimulator schreitet erst dann wieder fort, wenn eine Antwort in Form eines Ereignisses oder *look aheads* von dem Vorgänger eingetroffen ist.

Durch das beschriebene Verfahren wird einerseits ein hohes Nachrichtenaufkommen wie bei der Versendung von Nullnachrichten (siehe Abschnitt 3.1.2.2) vermieden und andererseits werden die Eigenschaften der zu simulierenden Netzwerke ausgenutzt, da mit Hilfe der Verzögerungen im Netzwerk die Simulation auch in einer Blockadesituation weiter vorangetrieben werden kann.

3.4. Nachrichtensysteme zur Verteilung der Simulation

Für den Austausch von Ereignissen und anderen Nachrichten zwischen Subsimulatoren stehen unterschiedliche Nachrichtensysteme zur Verfügung, deren Eignung für den Einsatz in der ereignisgetriebenen Simulation pulsverarbeitender neuronaler Netze in diesem Abschnitt untersucht werden. Zu den wichtigsten Anforderungen an ein Nachrichtensystem zur rechnerübergreifenden Verteilung der Ereignisse gehören eine weitgehende Plattformunabhängigkeit und eine niedrige Latenz sowie eine ausreichende Bandbreite. Durch eine Plattformunabhängigkeit lassen sich viele unterschiedliche Rechnersysteme unabhängig vom verwendeten Betriebssystem einbinden. Dadurch können bestehende Rechnerressourcen optimal genutzt werden. So kann auf den Einsatz von Spezialrechnern verzichtet werden. Dies gilt natürlich nur dann, wenn die Übertragungszeiten für Ereignisse kurz im Verhältnis zu deren Verarbeitungszeiten sind, also die Latenz der Übertragung niedrig ist. Bei großen Ereignismengen bestimmt außerdem die verfügbare Bandbreite die Transferzeiten. Darüber hinaus sollte mit dem Nachrichtensystem eine eventuell vorhandene Nebenläufigkeit zwischen der Übertragung und der Verarbeitung von Ereignissen ausgenutzt werden können.

Mit Hilfe der Transportprotokolle UDP[24] und TCP[25] lassen sich alle herkömmlichen Rechnersysteme koppeln, jedoch sind die Implementierungen dieser Protokolle nicht plattformunabhängig. Das Hauptproblem stellt hier die Repräsentation der Daten dar. Sie können im „little-endian“ oder im „big-endian“ Format ausgetauscht und verarbeitet werden. Zudem sind die Programmierschnittstellen der verschiedenen Implementierungen meist nicht kompatibel. Die Übertragung der Ereignisse mit Hilfe von UDP bzw. TCP wird jedoch mit in Betracht gezogen, da bei der Verwendung dieser Protokolle im Vergleich zu Programmbibliotheken aufgrund des geringeren Verwaltungsaufwands eine niedrigere Latenz zu erwarten ist. Für das UDP wurde einerseits die reine Transferleistung gemessen, andererseits die Leistung einer Bibliothek, die auf UDP basiert und einfache Mechanismen besitzt, die eine garantierte Übertragung der Ereignisse sicherstellt.

---

[24] **U**ser **D**atagram **P**rotocol

[25] **T**ransmission **C**ontrol **P**rotocol

Für den Einsatz im Simulationssystem kommt letztlich nur eine derart garantierte Nachrichtenübermittlung in Frage. Denn auch wenn mit dem Simulationssystem Netzwerke simuliert werden sollen, die für ihre Robustheit gegenüber unvollständigen Daten bzw. einer fehlerhaften Übertragung bekannt sind, so dient die Simulation in erster Linie dazu, das Verhalten solcher Netzwerke unter unverfälschten Bedingungen zu untersuchen.

Die Verwendung fertiger Programmbibliotheken, wie MPI[26], PVM[27], DCOM[28] oder CORBA[29] kann die Implementierung der Ereignisübermittlung vereinfachen, da in der Regel einheitliche Programmierschnittstellen auf allen von der Bibliothek unterstützten Plattformen vorliegen und zudem die Konvertierung der Datenformate für den Benutzer der Bibliothek transparent ist. Die Bibliotheken unterscheiden sich jedoch deutlich im Hinblick auf die von ihnen zur Verfügung gestellten Programmierschnittstellen und Funktionalitäten. PVM und MPI bieten eine C-Programmierschnittstelle, wohingegen DCOM und CORBA über objektorientierte Programmierschnittstellen verfügen.

PVM zielt insbesondere auf die Verknüpfung heterogener Rechnersysteme ab und unterstützt eine Vielzahl von unterschiedlichen Rechnerarchitekturen und Betriebssystemen. Dies umfasst einfache Workstations und PCs bis hin zu speziellen Parallelrechnern. Jedoch unterstützt PVM deutlich weniger Kommunikationsprimitive als MPI und bietet im Wesentlichen das Prinzip der Punkt-zu-Punkt-Kommunikation mit Sende- und Empfangsfunktionen an. Um spezielle Kommunikationsprimitive von parallelen Architekturen gezielt nutzen zu können, ist daher der Einsatz von MPI sinnvoller bzw. notwendig. MPI ist in der Version 1 noch auf homogene Rechnersysteme eingeschränkt. Der Standard ab Version 2 sieht jedoch die Ausweitung auf heterogene Rechnersysteme vor. Entsprechende Implementierungen des Standards auf unterschiedlichen Rechnerarchitekturen und Betriebssystemen entstehen zurzeit[30].

DCOM und CORBA zielen auf Client-Server-Anwendungen ab und bieten daher auch die Punkt-zu-Punkt-Kommunikation an. Sie stellen keine zusätzlichen Kommunikationsprimitive für spezielle Parallelrechner bereit. DCOM ist weitgehend auf die Windows-Betriebssysteme beschränkt und bietet somit eine proprietäre Lösung. CORBA–Implementierungen gibt es hingegen für eine sehr große Zahl von Rechnerarchitekturen und Betriebssystemen. Im Standard ist außerdem definiert wie Implementierungen verschiedener Hersteller interoperieren sollen, jedoch werden nicht alle Implementierungen dieser Anforderung gerecht. Durch ihre objektorientierten Programmierschnittstellen lassen sich DCOM und CORBA besonders homogen in die objektorientierte Architektur des Simulationssystems integrieren.

---

[26] **M**essage **P**assing **I**nterface

[27] **P**ure **V**irtual **M**achine

[28] **D**istributed **C**omponent **O**bject **M**odel

[29] **C**ommon **O**bject **R**equest **B**roker **A**rchitecture

[30] Stand der Implementierungen Heute (2001) entspricht dem MPI Standard in der Version 2 vom Juli 1997

3.4.1. TCP und UDP

Es ist notwendig, für jede Plattform[31], auf welcher der Simulator im Rahmen einer verteilten Simulation verwendet werden soll, eine eigene Implementierung für den Zugriff auf UDP bzw. TCP bereitzustellen. Diese Implementierung muss sowohl auf die Programmierschnittstelle der betreffenden Plattform angepasst sein als auch eventuell notwendige Konvertierungen der Daten bewerkstelligen. Für den Transfer über UDP muss außerdem ein Mechanismus bereitgestellt werden, der die Übertragung von Ereignissen garantiert.

Gegenüber TCP ist es mit UDP möglich so genannte Multicastnachrichten zu versenden. Beim Multicast kann ein Datenpaket an eine Untergruppe von Rechnern eines Netzwerksegments gleichzeitig verschickt werden. Für große pulsverarbeitende Netzwerke kann mit Hilfe dieses Verfahrens die zu transferierende Nachrichtenmenge reduziert werden. Die Multicastnachricht muss hierfür an mindestens zwei Nachfolger verschickt werden. Im Gegensatz dazu richtet sich eine Broadcast-Nachricht an alle in einem Netzwerksegment befindlichen Rechner gleichermaßen, was zu einer vergleichsweise geringeren Auslastung für eine Multicastverarbeitung führt. Idealerweise liegt dem Netzwerk eine Sterntopologie zugrunde, bei der die einzelnen Rechnersysteme eine direkte Verbindung zu einem Switch besitzen. Vernetzungen, bei denen die Rechnersysteme lediglich über einen Hub miteinander verbunden sind, können aus einer Multicast-Übermittlung keinen größeren Nutzen ziehen als unter der Verwendung der Broadcast-Übermittlung, weil keine Zwischenspeicherung der übermittelten Nachrichtenpakete erfolgt. Da in der Regel sehr unterschiedliche Netzwerktopologien vorliegen und in den meisten Fällen die Art der Vernetzung gar nicht bekannt ist, relativiert sich der praktische Nutzen der Multicastübermittlung. Hingegen stellt die Verwendung der Multicastübermittlung zusätzliche Anforderungen an das Nachrichtensystem des Simulators, da es in einem solchen Fall unterschiedliche Multicastgruppen verwalten muss. Dieser zusätzliche Verwaltungsaufwand macht schließlich den zu erwartenden Gewinn durch diese Übermittlungsart zunichte [Meissner2000].

Aufgrund der Einfachheit des UDP Protokolls könnte man erwarten, dass UDP eine geringere Latenz als TCP besitzt, was sich durch die Transfermessungen in Abschnitt 3.4.6 jedoch nicht bestätigte. Für die Messung der TCP Transferraten wurde zu Beginn der Messung ein Socket geöffnet, welcher die gesamte Zeit über geöffnet blieb.

Bei dem Vergleich der Nachrichtensysteme und der Bewertung der Messergebnisse ist dies zu berücksichtigen, insbesondere da die Zahl der Verbindungen, die gleichzeitig geöffnet sein können, auf den meisten Rechnersystemen begrenzt ist. Werden also alle Verbindungen wie in den Transfermessungen offen gehalten, so ist die Zahl der an der Simulation beteiligten Rechnersysteme durch diese Grenze beschränkt. Bei der Verwendung von UDP tritt dieses Problem nicht auf, da nur im Falle einer Kommunikation eine Verbindung geöffnet wird.

3.4.2. MPI

Auch wenn die vielen Kommunikationsprimitive, die das MPI bietet, im Simulationssystem nicht zum Tragen kommen, da es lediglich die einfache message-passing-Funktionalität vom darunter liegenden Nachrichtensystem benötigt, ist MPI eine gut geeignete Plattform zur Verteilung der Simulation. Dies gilt insbesondere, sofern spezielle

[31] z. B. SUN Solaris, Microsoft Windows oder Linux

parallele Rechnerarchitekturen verwendet werden sollen, mit denen eine besonders hohe Performance zu erreichen ist.

Steht hingegen ein Netzwerk mit Standardrechnern zur Verfügung, so ist das MPI für die Verteilung der Simulation weniger geeignet, da zur Zeit insbesondere heterogene Rechnernetze nicht ausreichend unterstützt werden. Auch wenn aus diesem Grunde von der Verwendung von MPI als Nachrichtensystem abgesehen wurde, würde eine konsequente Umsetzung des Standards in Implementierungen für die unterschiedlichen Rechnerarchitekturen und Betriebssysteme eine ideale Plattform für die Verteilung der Simulation schaffen, insbesondere, da dann die Leistungsfähigkeit von Parallelrechnern mit der Flexibilität heterogener Rechnernetzwerke verbunden werden könnte. Um die Implementierung des Simulationssystems so homogen wie möglich zu gestalten, wurde von einer Kombination mit anderen Nachrichtensystemen, wie z. B. CORBA, abgesehen. So könnte zwar bis zu einem bestimmten Grade die gewünschte Flexibilität auch erreicht werden, jedoch zu dem Preis einer deutlich schwerer wartbaren Applikation. Darüber hinaus würde eine solche Lösung die im Standard ohnehin vorgesehenen Eigenschaften nur unzureichend abbilden.

Für spätere Implementierung des Simulationssystems sollte jedoch die Verwendbarkeit der verfügbaren MPI-Implementierungen erneut geprüft werden. Die Architektur des Simulationssystems ist so ausgelegt, dass eine Integration von MPI als Nachrichtensystem leicht fallen sollte. Stünde eine MPI-Implementierung zur Verfügung, die eine objektorientierte Programmierschnittstelle bereitstellt, so würde die Integration noch unkomplizierter sein.

Im Januar 2000 wurde das erste Protokoll zum Interoperablen-Message-Passing-Interface (IMPI) abgefasst, welches die Interoperabilität von MPI-Implementierungen zum Ziel hat. Mit IMPI soll ein standardisiertes Protokoll für die Kommunikation unterschiedlicher MPI-Implementierungen geschaffen werden. Implementierungen unterschiedlicher Hersteller, die diesen Protokollstandard erfüllen, sind dann in der Lage zu interoperieren. Dieser Ansatz entspricht den bereits in CORBA definierten und umgesetzten Lösungen zur Interoperabilität unterschiedlicher CORBA-Implementierungen. Zu IMPI konforme MPI-Implementierungen bieten damit in Zukunft die Vorteile beider Systeme: Einerseits die elegante und standardisierte Unterstützung heterogener Umgebungen, andererseits die besonders effiziente Ausnutzung von parallelen Rechnerarchitekturen. Bei entsprechender Umsetzung der Standards kann schließlich eine auf IMPI basierende Applikation sowohl auf einem Netz aus Workstations als auch auf einem Parallelrechner ausgeführt werden, ohne dass Anpassungen in der Implementierung vorzunehmen wären. Entsprechend effiziente IMPI Implementierungen vorausgesetzt, wäre dieser Standard in Zukunft für die Verwendung als Nachrichtensystem im Simulator zu favorisieren.

3.4.3. PVM

PVM [GeiBegJia1994] ist eine C-Bibliothek, welche auf einige Rechnersysteme portiert wurde. Die Bibliothek stellt neben den Funktionen zum Nachrichtentransfer (send und receive) auch Funktionen für den Start von Prozessen auf entfernten Rechnern bereit. Zudem wird eine automatische Lastverteilung unterstützt, so dass es für den Nutzer transparent ist, auf welchem Rechner entsprechende Prozesse verarbeitet werden. PVM bietet somit die Möglichkeit, einen Verbund von Rechnern als eine virtuelle Maschine erscheinen zu lassen. Die Bibliotheksfunktionen bieten jedoch auch die Möglichkeit, die Verteilung der Prozesse zu kontrollieren und lediglich eine explizite

Kommunikation zwischen diesen Prozessen stattfinden zu lassen. PVM unterstützt eine Punkt-zu-Punkt-Kommunikation. Weitere Kommunikationsprimitive, wie sie in Parallelrechnern zur Verfügung stehen, werden nur unzureichend oder gar nicht abgebildet.

Auch wenn einige Implementierungen für parallele Rechnerarchitekturen existieren, so sind die PVM-Implementierungen meist nicht interoperabel und können daher nicht miteinander gekoppelt werden. Ein heterogener Betrieb von Parallelrechnern und Workstations ist somit in den meisten Fällen auszuschließen.

Da zu Beginn der Entwicklung des SPIKELAB Simulationssystems PVM, im Gegensatz zu CORBA oder MPI, eine ausgereifte und leicht verfügbare Implementierung bereitstellte, die zumindest die Einbindung von PCs und Workstations ermöglichte, welche entweder mit Solaris, Linux oder Windows NT betrieben wurden, entschloss sich der Autor die erste Version des verteilten Simulators mit PVM als Nachrichtensystem zu realisieren. Eine Beschreibung der Implementierung findet sich in [Schild1999].

Nachdem die CORBA-Implementierungen einen entsprechenden erreicht hatten, lösten diese PVM als Nachrichtensystem ab. Dies begründet sich einerseits in der Performance, wie dies den folgenden Messungen entnommen werden kann, und andererseits in der leichteren Handhabbarkeit der CORBA-Implementierung.

3.4.4. CORBA

CORBA ist eine standardisierte Sammlung von Diensten zur verteilten Kommunikation von Objekten einer Applikation. Den Kern einer CORBA-Implementierung bildet der Object Request Broker (ORB), ein Dienst, der die Aufrufe zwischen den Objekten einer Applikation vermittelt und, falls notwendig, auch entsprechende Objekte mit Hilfe anderer Dienste erzeugt, die ebenso im CORBA-Standard definiert sind. Der Standard fordert dabei eine strenge Trennung des Interfaces eines Objektes von dessen Implementierung. Für die Beschreibung des Interfaces existiert eine eigene Sprache, die sehr stark an C++ angelehnt ist. Durch die strenge Trennung von Interface und Implementierung und die Verwendung einer standardisierten Interfacebeschreibung gelingt es, die Implementierung des Objektes soweit zu entkoppeln, dass es weder von Bedeutung ist, wie und in welcher Sprache das Objekt letztlich implementiert ist, noch wo es ausgeführt wird. Für die verwendete Programmiersprache muss jedoch eine entsprechende Sprachbindung vom ORB bereitgestellt werden. In der Regel werden Sprachbindungen für C++ und Java von allen ORBs unterstützt.

CORBA sieht eine Punkt-zu-Punkt-Kommunikation in zwei verschiedenen Varianten vor: blockierend mit garantierter Übermittlung und nicht blockierend ohne garantierte Übermittlung. Da die Übermittlung der Nachrichten bei der nicht blockierenden Variante nicht garantiert ist und zudem die Ausführung eines so genannten „oneway“-Aufrufs nicht weiter im Standard spezifiziert ist, sogar in absehbarer Zeit aus diesem entfallen soll, verbleibt die Verwendung blockierender Aufrufe. Die Kommunikation wird dabei nicht durch den Aufruf bestimmter Bibliotheksfunktionen realisiert, sondern es werden direkt die Methoden eines Objektes aufgerufen, wodurch es innerhalb der Implementierung transparent ist, ob das Objekt, dessen Methode aufgerufen wird, lokal oder auf einem entfernten Rechner existiert.

Obwohl CORBA für den Bereich der Client-Server-Applikationen konzipiert worden ist, zeigen einige Implementierungen eine zu Protokollen wie TCP/IP vergleichbare Performance, als auch eine relativ niedrige Latenz. Neben der Performance der

Implementierung spielt bei der Auswahl eines Nachrichtensystems auch der Aufwand für seinen Einsatz eine wesentliche Rolle. Hierbei spielen neben der eigentlichen Implementierung Zusatztools ebenso eine Rolle, wie auch die Dokumentation und Verständlichkeit der Programmierschnittstelle. Zieht man diese Aspekte mit in Betracht, so stellt CORBA im Vergleich zu PVM die einfachere Plattform zur Verteilung einer Simulation dar, insbesondere da der neben der eigentlichen Nachrichtenübermittlung zu implementierende Teil für die Verwaltung einfacher ausfällt. Darüber hinaus resultiert aus der Verwendung von CORBA eine homogene objektorientierte Implementierung, welche zu einem wesentlich besser wartbaren System führt. In [Meissner2000] ist eine detaillierte Abhandlung zur Auswahl einer geeigneten CORBA Implementierung zu finden.

3.4.5. COM/DCOM

Die Konzepte, welche COM/DCOM zugrunde liegen, weisen viele Gemeinsamkeiten mit denen von CORBA auf. Daher ist im CORBA-Standard mittlerweile auch die Anbindung von DCOM- bzw. COM-Komponenten definiert. Bei genauerer Betrachtung bietet DCOM gegenüber CORBA keine Vorteile, insbesondere da es sich um eine proprietäre Lösung handelt. Erstens führt es zu einer Bindung an ein Betriebssystem und zweitens stellt es einen Kostenfaktor dar, denn bei DCOM kann nicht wie bei CORBA auf kostenfreie Implementierungen zurückgegriffen werden.

DCOM ermöglicht genauso wie CORBA den transparenten Zugriff auf Objekte innerhalb einer Implementierung durch die strenge Trennung von Interface und Implementierung. Hierfür wird genauso wie in CORBA eine standardisierte Interfacebeschreibung verwendet. Eine Interoperabilität mit anderen Implementierungen ist hingegen nicht in DCOM verankert, da es sich um eine Lösung eines einzelnen Herstellers handelt.

3.4.6. Leistungsvergleich der Nachrichtensysteme

PVM, CORBA und die einfache TCP- bzw. UDP-Übertragung wurden im Rahmen dieser Arbeit einem Leistungsvergleich unterzogen, wobei die für eine Simulation charakteristische Auslastung des Nachrichtensystems berücksichtigt wurde, indem die für den Test verwendeten Nachrichtengrößen einem Vielfachen der Größe eines Ereignisses entsprachen.

Die Systeme wurden bezüglich ihrer Latenz, Bandbreite und - sofern möglich - des zeitlichen Aufwands für das Konvertieren[32] der Daten von der vorliegenden, nativen oder objektorientierten Datenstruktur in das vom Nachrichtensystem verwendete plattformunabhängige Datenformat[33] untersucht. Im Simulationssystem wurden letztlich PVM und CORBA integriert und bewertet, siehe auch [Schild1999] und [Meissner2000].

Die Übertragungszeiten und Bandbreiten wurden mit Hilfe einer einfachen Echoapplikation gemessen, die mehrfach ein Datenpaket wachsender Größe zum Zielrechner schickt und von diesem als Antwort ein ebenso großes Datenpaket zurück erhält. Die angegebenen Zeiten gelten deshalb für zwei Sende- und Empfangsvorgänge, wohingegen bei der Angabe der Bandbreite bereits berücksichtigt ist, dass in der gegebenen Zeit zweimal die Paketgröße transferiert wurde. Jedes Datenpaket wurde jeweils 100 Mal

[32] Hierfür wird häufig der englischsprachige Begriff *marshalling* verwendet.

[33] In der Regel findet eine Konvertierung in die so genannte *Network-Byte-Order* statt.

hin- und zurückgeschickt. Nach den 100 Wiederholungen wurde die Paketgröße in Schritten von 16 Byte erhöht, was der Größe eines Ereignisses entspricht. Die Messreihen wurden mehrfach mit 4 Byte und mit 16 Byte bis 65536 Byte durchgeführt. Bei manchen Messungen wurde bei einer kleineren Datenmenge abgebrochen, da entweder bereits eine Sättigung der Bandbreite eingetreten war oder eine Messung oberhalb einer bestimmten Paketgröße nicht mehr möglich war (z. B. einfaches UDP). Auf den Solarissystemen und den WindowsNT-Systemen wurden die gleichen Quellen verwendet, wobei plattformspezifische Stellen des Codes mit Präprozessordirektiven versehen wurden. Bei TCP-Messungen ohne weitere Angaben konnte das System Datenpakete im Duplexmodus austauschen[34], wobei bei der blockierenden Variante immer gewartet wurde, bis das gesamte Paket auf einer Seite eingetroffen war, bevor es anschließend zurückgeschickt wurde. Die PVM-Implementierung nutzt im Hintergrund immer dieses Verfahren, da die Nachrichten gepuffert werden. Dass diese Pufferung jedoch auch ihre Nachteile besitzt, kann den Messungen entnommen werden, bei denen das Packen der Daten in den PVM-Puffer mit berücksichtigt wurde. Es handelt sich dabei um die Messungen, die mit dem Kürzel P versehen sind.[35] Bei diesen Messungen wurde vor jedem Versenden die Funktion zum Packen des Buffers aufgerufen. Bei den anderen Messungen (Messungen, die mit dem Kürzel K.P. versehen sind.) wurde die Funktion zum Packen des Puffers nur einmal für jede Paketgröße aufgerufen und floss nicht in die Zeitmessung mit ein. Bedingt durch die Implementierung von CORBA wurden die Messungen grundsätzlich inklusive der Datenkonvertierung durchgeführt. CORBA kann bei der blockierenden Übertragung aus dem Duplexmodus keinen Nutzen ziehen. Auch die einfache UDP-Übertragung wurde so implementiert, dass stets ein kompletter Transfer des Datenpakets erfolgte, bevor es zurückgeschickt wurde.

Die Messungen zwischen Workstations der Firma SUN mit dem Betriebssystem Solaris wurden in einem Rechnerpool durchgeführt, in dem alle Rechner über einen 100 MBit Switch innerhalb eines LANs miteinander vernetzt waren. Zudem wurden die Messungen zu Zeiten geringer Netzauslastung durchgeführt. Die Messungen mit PCs, auf denen das Betriebssystem Windows NT 4.0 verwendet wurde, fanden an einem eigens für diese Messungen abgestellten 100MBit Switch statt, so dass dort ein weiterer Netzverkehr ausgeschlossen werden konnte.

Bei den Messungen zwischen PCs (Windows NT) und SUN-Workstations (Solaris) wurde auf ein 100 MBit LAN zurückgegriffen, welches die genannten Rechnersysteme über jeweils drei 100 MBit Switches verknüpft. Auch diese Messungen wurden zu Zeiten geringer Netzauslastung durchgeführt. Dennoch ist bei dem Vergleich der Ergebnisse zu berücksichtigen, dass die Übertragungszeit und insbesondere die Latenz durch die zusätzlichen Switches grundsätzlich etwas höher ausfallen.

Tabelle 4 zeigt die geringste, die mittlere und die größte gemessene Latenz der verschiedenen Übertragungen in Sekunden für die Versendung und den anschließenden Empfang eines 4 Byte Datenpakets. Die Tabelle ist aufsteigend sortiert.

---

[34] TCP teilt automatisch größere Datenpakete in kleinere auf, die dann unabhängig voneinander transferiert werden können, so dass der Sender einer Echoapplikation immer noch senden kann, während der Empfänger bereits empfangene Pakete unter Ausnutzung der Duplexfähigkeit wieder zurücksenden kann. Die blockierende Implementierung hebelt diesen Mechanismus aus, indem beim Empfänger gewartet wird, bis das gesamte Datenpaket eingetroffen ist, bevor es zurückgesendet wird.

[35] Das Packen des PVM-Buffers beinhaltet gleichzeitig die Datenkonvertierung in die *Network Byte Order*

Erwartungsgemäß weisen die Protokolle TCP und UDP die geringste Latenz auf. Deutlich fällt die große Latenz der PVM Übertragung zwischen PC - SUN und PC - PC auf. Sie liegt fast um den Faktor 100 höher als bei der Übertragung mittels TCP zwischen PCs. Dies scheint ein Problem der Windows-Implementierung von PVM zu sein, denn bei der Übertragung zwischen SUN-Workstations tritt eine deutlich geringere Latenz auf, die vergleichbar mit der etwas besseren Latenz einer CORBA-Übertragung auf der gleichen Plattform ist. Schließlich liegen die Messwerte für die plattformübergreifende Übertragung grundsätzlich höher, was zum Teil auf die ungünstigere Vernetzung und damit auch potentiell höhere Wahrscheinlichkeit von Störungen zurückzuführen ist. Aufgrund dieser Messung lassen sich vor allem zwei Schlüsse ziehen:

1. Sobald eine zusätzliche Verarbeitung des Nachrichtenstroms notwendig wird, steigt die Latenz merklich an. Vergleicht man den Messwert für die UDP_LIB-Übertragung zwischen SUN-Workstations, bei der lediglich die garantierte Übertragung sichergestellt ist, mit der entsprechenden TCP-Übertragung auf der gleichen Plattform, so erweist sich die TCP-Implementierung als überlegen.
2. Bei der Verwendung der Nachrichtensysteme PVM und CORBA ist CORBA wegen der grundsätzlich besseren Performance vorzuziehen, zumal es den Schwachpunkt in der auf Windows basierenden PVM-Implementierung gibt.

Die etwas besseren Ergebnisse der Messungen für reine PC-Kommunikation können darauf zurückgeführt werden, dass die Messungen mit den PCs über einen Switch erfolgte, der lediglich diese beiden an der Messung beteiligten Rechner verband, während die Messungen mit den SUN-Workstations über einen Switch erfolgte, der sich in einem LAN befand.

Übertr.	Verbind.	N.B. / B. P. / KP.	geringste Latenz (ms)	mittlere Latenz (ms)	größte Latenz (ms)
TCP	PC_PC	N.B.	0,136	0,141	0,143
UDP	PC_PC	N.B.	0,134	0,141	0,148
TCP	PC_PC	B.	0,142	0,145	0,153
UDP	SUN_SUN	B.	0,186	0,188	0,192
TCP	SUN_SUN	N.B.	0,195	0,197	0,202
TCP	SUN_SUN	B.	0,198	0,199	0,202
CORBA	PC_PC	B.	0,201	0,212	0,285
UDP_LIB	SUN_SUN	B.	0,212	0,230	0,235
CORBA	SUN_SUN	B.	0,370	0,375	0,407
PVM	SUN_SUN	K.P.	0,387	0,394	0,416
PVM	SUN_SUN	P.	0,453	0,456	0,460
TCP	PC_SUN	B.	0,283	0,741	1,037
UDP	SUN_PC	B.	0,657	0,912	1,047
CORBA	PC_SUN	B.	1,124	1,155	1,177
PVM	SUN_PC	K.P.	9,448	10,125	11,42
PVM	PC_PC	K.P.	10,068	10,798	11,32

Tabelle 4: Latenz verschiedener Übertragungen in Millisekunden (Paketgröße jeweils 4 Byte)[36]

[36]Abkürzungen B = blockierend, K.P. = kein Packen P. = Packen (Erläuterungen siehe Text)

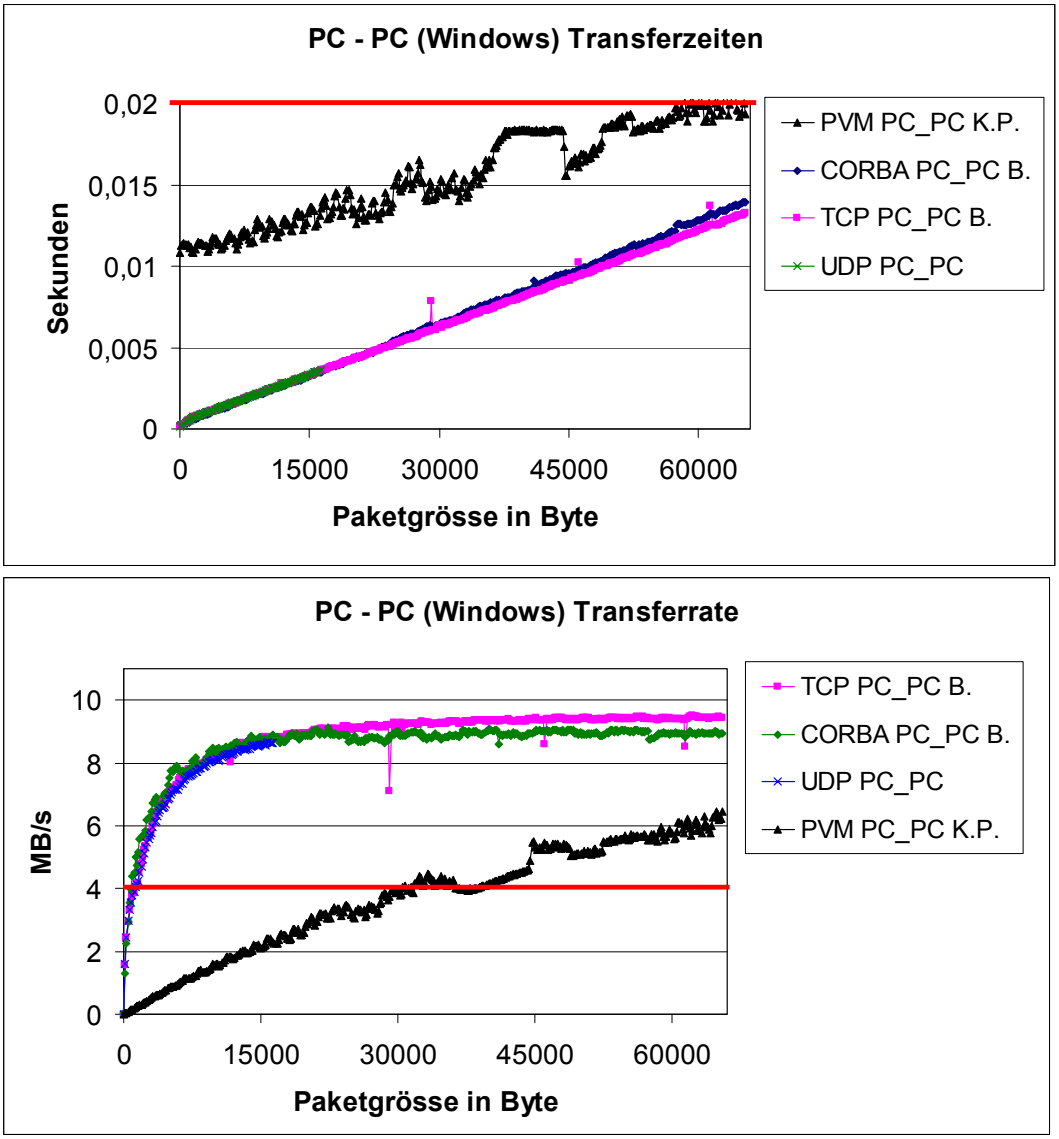


Abbildung 29: Transferzeiten und –raten für PC-PC (Windows) Kommunikation.

Die Transferzeiten und -raten sind in Abbildung 29, Abbildung 30 und Abbildung 31 wiedergegeben. Die ersten drei Diagramme zeigen die Transferzeiten und -raten getrennt nach Plattformen. Da die Diagramme auf der Abszisse unterschiedlich skaliert sind, ist in allen Diagrammen der Wert 20ms bzw. 4MB/s rot hervorgehoben. Es fällt die deutlich ungünstigere Transferzeit von PVM beim Transfer zwischen WindowsNT-PCs (Abbildung 29) auf, die sich bei den Messungen zwischen den Plattformen SUN-Solaris und WindowsNT-PC (Abbildung 31) aufgrund der insgesamt höheren Transferzeiten nur bei kleineren Datenmengen bemerkbar macht.

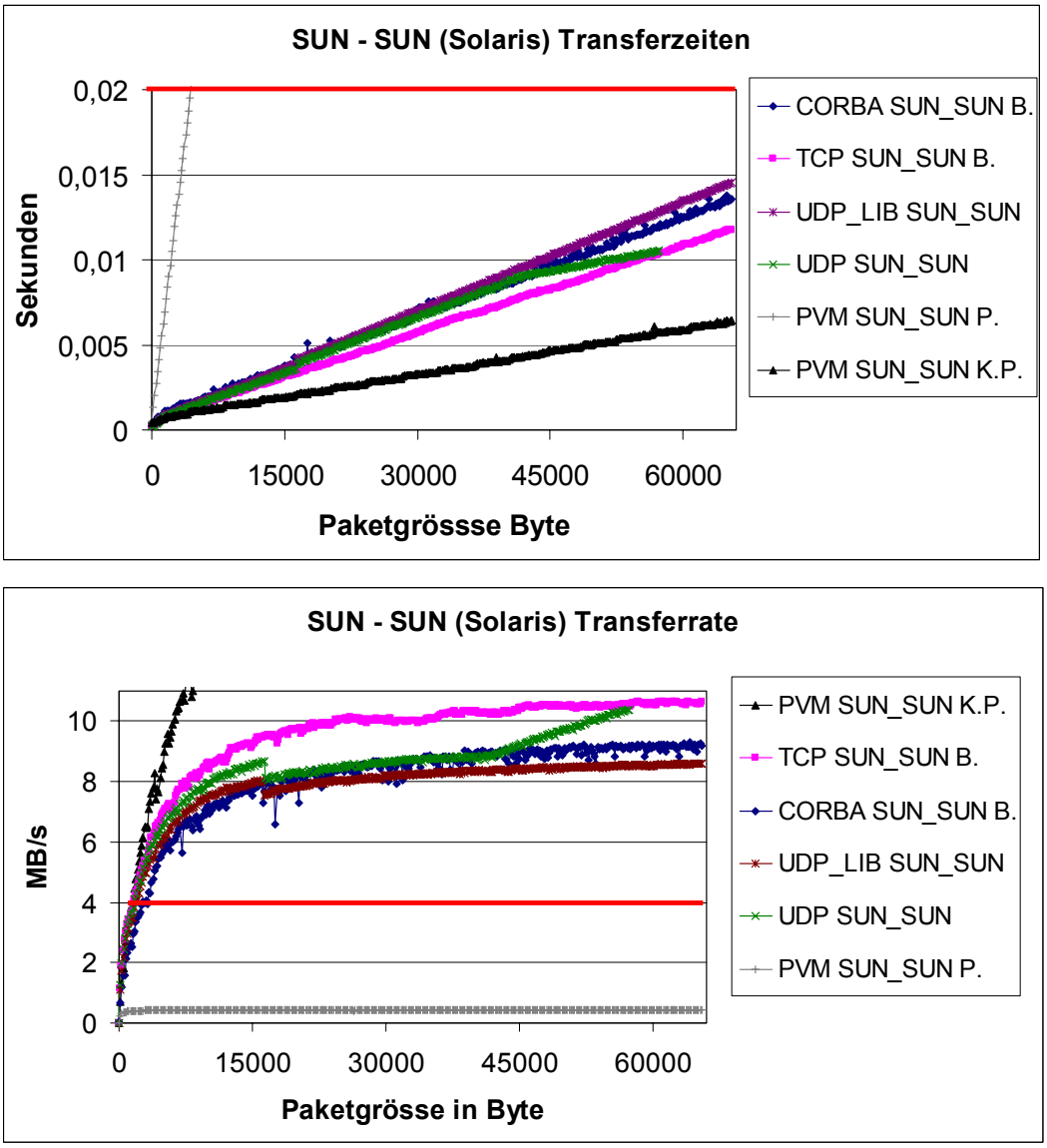


Abbildung 30: Transferzeiten und -raten für SUN - SUN (Solaris) Kommunikation

Die wesentlich kürzere Transferzeit für eine PVM-Übertragung und eine entsprechend hohe Transferrate zwischen SUN-Workstations ist auf die in PVM verwendete Pufferung zurückzuführen. Dadurch kann der Duplexmodus ausgenutzt werden, womit theoretisch die doppelte Datenrate erreicht werden könnte (siehe auch Abbildung 32). Wird jedoch das Packen der Daten mit berücksichtigt, so wird die Transferrate durch den Packvorgang dominiert und erreicht weit geringere Werte als bei den anderen Nachrichtensystemen (graue Linie in Abbildung 30).

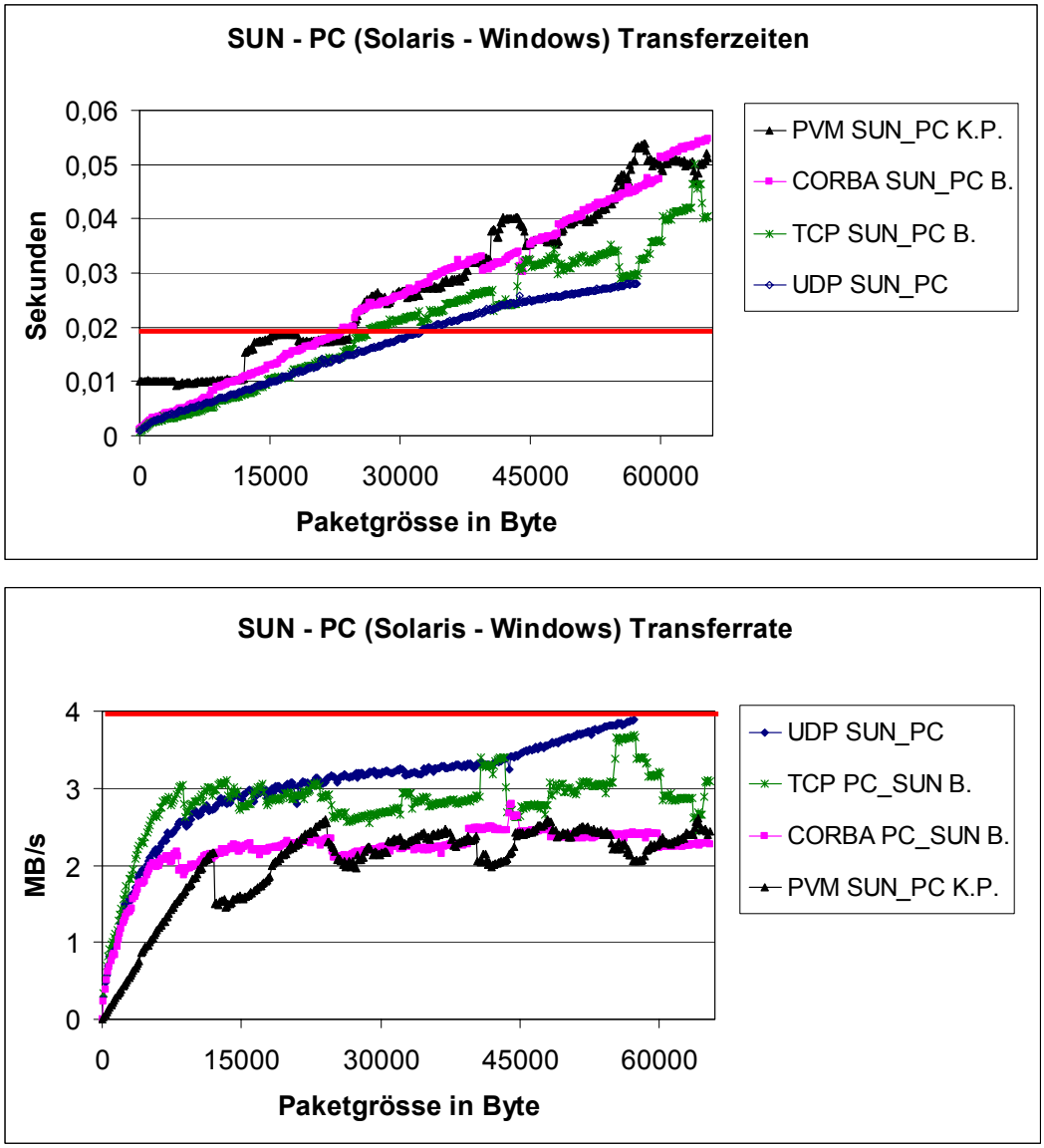


Abbildung 31: Transferzeiten und -raten für SUN - PC (Solaris - Windows) Kommunikation

Trotz der grundsätzlich langsameren Übertragung zwischen den Plattformen SUN (Solaris) und PC (Windows) ist der Offset von 10ms bei der PC(Windows)-Übertragung deutlich in Abbildung 31 links zu erkennen. Des Weiteren fällt auf, dass der Abstand zwischen einer TCP Übertragung und eine CORBA Übertragung bei einer PC – PC Verbindung vernachlässigbar ist, jedoch bei einer SUN – SUN Verbindung für größere Datenmengen ca. 16% beträgt und Bei SUN – PC Verbindungen sogar ca. 25%. Dieser Unterschied macht sich ab einer Paketgröße von ca. acht Kilobyte bemerkbar.

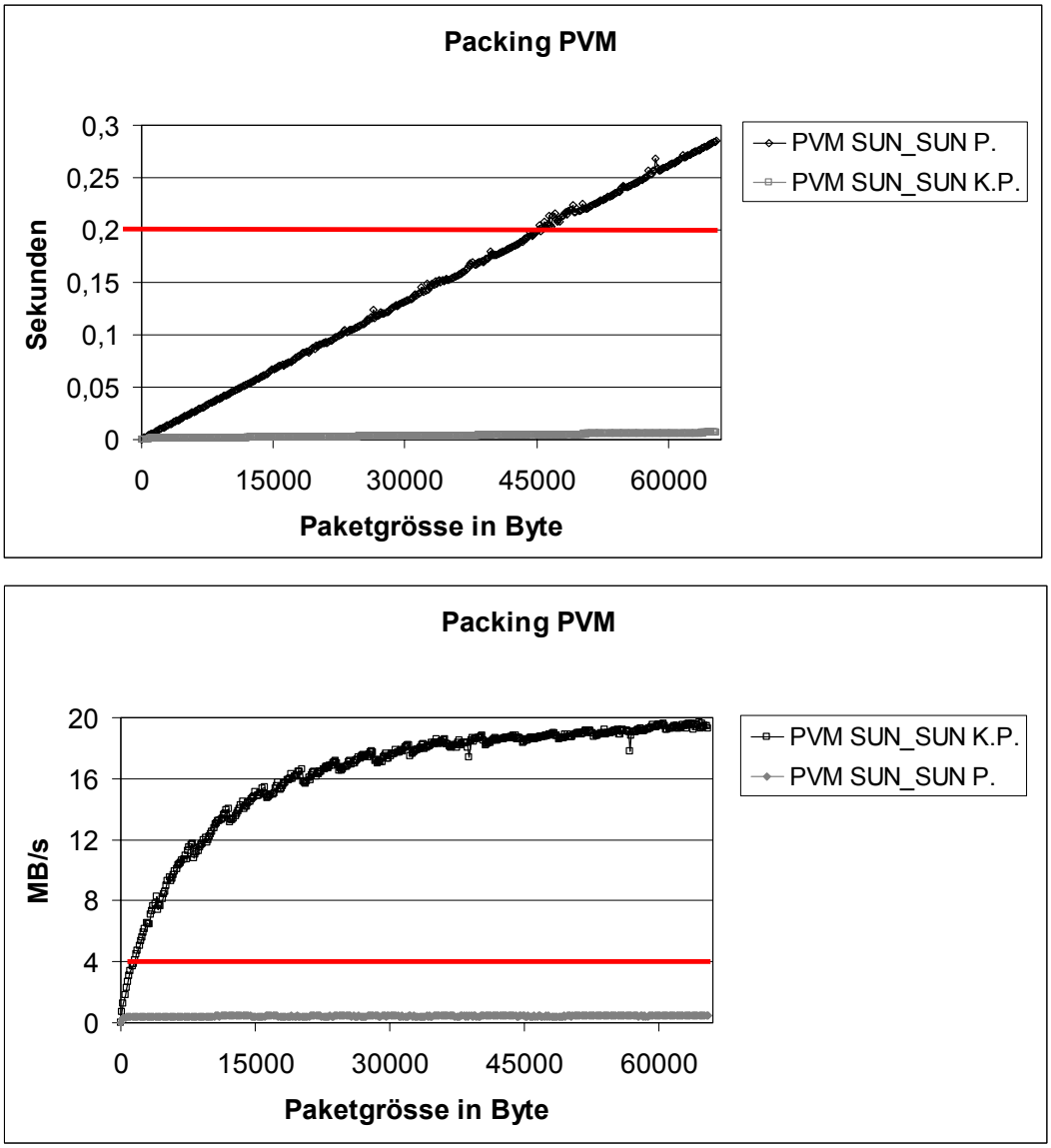


Abbildung 32: Einfluss des *Packing* bei PVM

3.4.7. Auswahl des Nachrichtensystems

Wie schon zuvor im Abschnitt über PVM zu lesen war, wurde für die erste Implementierung des Simulationssystems PVM verwendet, da CORBA Implementierungen zum Zeitpunkt der Designentscheidung noch nicht weit genug entwickelt waren. Im Verlauf der Arbeit bot sich dann jedoch die Möglichkeit, auch eine Implementierung in CORBA zu realisieren. Sie löste die PVM basierte Variante ab. Dabei wurden wesentliche Aspekte der Schnittstellen zum Nachrichtensystem modifiziert. Die Modifikationen wurden insbesondere vorgenommen, damit sich in Zukunft das Nachrichtensystem leichter austauschen lässt, da einige Probleme bei dem Wechsel von PVM zu CORBA erst dann offensichtlich wurden. Damit ist die jetzt vorliegende und hier beschriebene Implementierung jedoch nicht mehr kompatibel mit der ursprünglichen PVM-Implementierung, stellt aber ein deutlich besseres Interface für die Einbindung anderer Nachrichtensysteme, wie zum Beispiel MPI, zur Verfügung. Zudem ist den Messungen zu entnehmen, dass CORBA die bessere Performance bietet, vor allem wenn die Simulation mit Hilfe eines Netzwerks von WindowsNT-PCs durchgeführt werden soll. Darüber hinaus kann

bei CORBA auf zusätzliche Tools, wie zum Beispiel den RSH-Deamon für die auf Windows basierende PVM-Implementierung, verzichtet werden. Auch fällt die Verwaltung[37] der Simulation im heterogenen Umfeld mit CORBA deutlich leichter.

3.5. Verwandte Arbeiten und Simulationssysteme

Es gibt eine Vielzahl von Simulationssystemen für pulsverarbeitende neuronale Netze, die auf einer Zeitscheibensimulation basieren. Bis zum jetzigen Zeitpunkt existiert jedoch nur ein Simulationssystem, welches ein ereignisgetriebenes Verfahren zur Simulation pulsverarbeitender Netze einsetzt. Es handelt sich dabei um den Simulator SPIKE von Lloyd Watts [Watts1994].

3.5.1. Zeitscheibensimulation

3.5.1.1. GENESIS und PGENESIS

Unter den Zeitscheibensimulatoren für pulsverarbeitende neuronale Netze sticht ein Simulationssystem aufgrund seines breiten Einsatzes in verschiedenen Forschungsvorhaben hervor. Dies ist der Simulator GENESIS, welcher insbesondere zur Simulation von Abschnitts-Modellen ausgelegt ist. Der Aufbau und die Simulation von Modellen, die durch entsprechende Abschnitte (*compartments*) beschrieben werden, können dabei graphisch oder über ein Skript erfolgen. GENESIS ermöglicht die Simulation sowohl kleiner als auch mittlerer Netzwerkgrößen und bietet die Möglichkeit, mit verschiedenen Zeitauflösungen zu simulieren.

PGENESIS ist eine parallele Implementierung des GENESIS-Simulators. Die Parallelisierung, respektive die Verteilung der Simulation wird dabei durch die Bereitstellung von Skripten für jeden an der Simulation beteiligten Rechner realisiert, wobei jedes dieser Skripte den jeweiligen Teil des Netzwerks enthält, der auf dem betreffenden Rechner simuliert werden soll. Die Synchronisation und die Blockadefreiheit muss der Benutzer gewährleisten, indem er entsprechende Skripte auslegt, wobei er sich der Synchronisationsmechanismen bedienen kann, die von PGENESIS bereitgestellt werden.

3.5.2. Spezialisierte Zeitscheibensimulation

Bei der Zeitscheibensimulation gibt es einige Beispiele für spezialisierte Simulationsverfahren, bei denen, um die Simulation zu optimieren, Parameter des simulierenden Systems fixiert werden (z. B. feste Zeitauflösung von 1 ms), oder Techniken ereignisgetriebener Verfahren wie Ereignislisten integriert werden [Wolff2001]. Auch lässt sich der Speicherbedarf verringern, indem Regelmäßigkeiten in der Vernetzung für Vereinfachungen genutzt werden[Delorme1998].

Stellvertretend sei hier der Simulator SpikeNet [Delorme1998] kurz beschrieben und bewertet.

---

[37] Zur Verwaltung gehört z. B. das Starten von Prozessen (Subsimulatoren) auf entfernten Rechnern und der Aufbau einer initialen (plattformübergreifenden) Kommunikation auch unter Verwendung sicherer Kommunikationsmethoden, wie z. B. ssh.

3.5.2.1. SpikeNet

SpikeNet [Delorme1998] implementiert ähnlich wie die Simulation von [Wolff2001] eine Mischform zwischen der Zeitscheibensimulation und der ereignisgetriebenen Simulation, da hier Listen mit den aktiven Neuronen geführt werden, diese jedoch in äquidistanten Schritten geprüft und ausgewertet werden. Die Simulation ist wie in [Wolff2001] daraufhin optimiert, dass die zeitliche Auflösung 1 ms beträgt. In [Delorme1998] ist angegeben, dass das Verfahren von der Auflösung nicht abhänge. Dies muss aufgrund des gewählten Verfahrens in Frage zu gestellt werden, zumal der Autor entsprechende Nachweise schuldig blieb. Weiterhin wird in allen untersuchten Beispielen (Gesichtserkennung) von einer Aktivität ausgegangen, die geringer ist als ein Puls pro Neuron und Sekunde. Verzögerungen sind in den angeführten Beispielen nicht enthalten. Nach Aussage des Autors sein sie einfach in der Simulation zu berücksichtigen. Aus den Ausführungen von Abschnitt 3.2.5 wird jedoch ersichtlich, dass dies weitere Maßnahmen erfordern würde. Somit ist zu erwarten, dass einige der verwendeten Vereinfachungen im Simulationsablauf wegfallen müssten und damit eine deutlich geringere Leistungsfähigkeit erreicht würde.

Laut Angaben des Autors sind bis zu 32 Mio. Neuronen und 245 Milliarden Verbindungen mit dem vorgestellten Simulationsverfahren auf einem Macintosh PowerPC realisierbar. Derartige Netzwerkgrößen werden nur dann erreicht, wenn das Gros aller Verbindungen identisch ist und deshalb gemeinsam genutzt werden kann (Fachterminus: *shared weights*). In einem solchen Falle verwenden viele Neuronen die gleichen Gewichte. Bei der genannten Größenordung müssen die Zielneuronen zudem berechnet werden und können nicht explizit abgespeichert werden (keine Inputnummer oder ähnliches). Dies erfordert einerseits, dass die Netzstruktur sehr regulär ist, und andererseits, dass auf individuell modifizierbare Gewichte verzichtet wird.

SpikeNet kann, da es auf einen bestimmten Anwendungsfall spezialisierte wurde, in seinem beschränkten Anwendungsfeld jedoch eine sehr gute Leistungen erzielen, dies gelang insbesondere, weil die gewählten Einschränkungen konsequent in eine speicher- und vor allem cache-optimierte Implementierung umgesetzt wurden [Delorme1998].

3.5.3. Ereignisgetriebene Simulationsverfahren

3.5.3.1. SPIKE

Das Simulationssystem SPIKE von Lloyd Watts [Watts1994], welches auf ein bestimmtes Neuronenmodell hin implementiert und optimiert wurde, stellt eine sequentielle Simulation bereit, für die keine Verteilung vorgesehen ist. Das verwendete Neuronenmodell integriert eingehende Pulse von Vorgängerneuronen für jeden Eingang linear und summiert die Beiträge der einzelnen Eingänge zu einem Summenpotential und sendet selbst einen Puls aus, wenn das Summenpotential einen festen Schwellenwert erreicht. Für die einzelnen Eingänge können Gewichte definiert werden, wie auch Wirkdauern, die festlegen, wie lange ein eingehender Puls auf den betreffenden Eingang einwirkt. Außerdem kann noch ein Sättigungswert für jeden Eingang definiert werden, der den Beitrag eines Eingangs auf das Summenpotential beschränkt. Da das betrachtete System stückweise lineare Verläufe aufweist, liegt es nahe, dass neben dem Eintreffen eines Pulses, das heißt der Auslösung eines Rechteckpulses mit der für den betreffenden Eingang festgelegten Wirkdauer und durch das korrespondierende Gewicht festgelegten Höhe, auch das Ende eines solchen Pulses als Ereignis modelliert wird. Durch eine derartige Modellierung müssen lediglich die linearen Verläufe zwischen den Ereignissen

berechnet werden und dabei muss überprüft werden, ob der Schwellenwert für das aktuell betroffene Neuron überschritten wird. Auf der anderen Seite ist damit das Verfahren auf das beschriebene Neuronenmodell festgelegt, d.h. der Simulationskern ist stark von diesem Modell abhängig. Die Modellierung bedingt zudem, dass der Prozess, welcher die Zustandsänderungen der Neuronen berechnet, mit dem Simulationskern eine Einheit bildet.

Kapitel 4 - Das SPIKELAB Simulationssystem

4.1. Das Gesamtsystem im Überblick – Hard- und Software

Das Simulationssystem SPIKELAB setzt sich aus mehreren Komponenten zusammen, die alle gemeinsam dem Ziel einer effizienten Simulation pulsverarbeitender neuronaler Netze dienen. Die Eingabe und Visualisierung von Netzwerken von Neuronen sowie Kombinationen dieser Netzwerke zu Topologien werden durch eine graphische Benutzeroberfläche unterstützt, welche mit Hilfe der Microsoft Foundation Classes (MFC) implementiert ist und somit plattformabhängig auf Rechnern mit einem Windows Betriebssystem ausführbar ist [Lohwasser1999]. Nutz man kommerzielle Portierungen der MFC kann die Oberfläche auch auf andere Betriebssysteme portiert werden. Zur detaillierten Visualisierung der Netzwerke ist ein Modul, basierend auf dem OpenGL-Standard, zur dreidimensionalen Darstellung der Topologien in die Oberfläche integriert [Troidl2000]. Für die Simulation großer und unregulärer Topologien kann eine automatische Partitionierung eingesetzt werden. Diese basiert auf Algorithmen zur Graphenpartitionierung, welche auf die Bedürfnisse von pulsverarbeitenden neuronalen Netzen angepasst wurden [Schulz1999].

Das Zentrum des ganzen Systems bildet der Simulatorkern, welcher portabel implementiert ist. So kann er auf die Betriebssysteme Windows, Solaris und Linux übersetzt werden. Die Schnittstellen zur Verteilung der Simulation sind so konzipiert, dass verschiedene Systeme für den Nachrichtenaustausch verwendet werden können. Die Verteilung wurde sowohl mit Hilfe von PVM [Schild1999] als auch CORBA [Meissner2000] implementiert. Der Einsatz weiterer Systeme, wie z. B. MPI, ist möglich. Zum analysieren der Simulationsergebnisse diente im Rahmen von SPIKELAB ein weiteres Modul implementiert, welches die Analyse von zeitlichen Korrelationen innerhalb von Neuronengruppen und die Visualisierung von Pulsfolgen ausgewählter Neuronen ermöglicht [Alexander1999].

Schließlich beinhaltet SPIKELAB die flexible Hardwareplattform RACER, mit deren Hilfe sich die Simulation beschleunigen lässt und pulserzeugende Sensoren eingebunden werden können. Sie erlaubt es daher auch pulsverarbeitende und –erzeugende analoge wie digitale Hardware anzubinden.

4.1.1. Topologie – Graphische Oberfläche - 3D Visualisierung

Mit Hilfe der graphischen Oberfläche werden neuronale Netzwerke in einer Hierarchie angeordnet und verwaltet. Die Gesamtheit der hierarchisch angeordneten Netzwerke ist die Topologie. Zur Visualisierung der Hierarchie wird ein Baum eingesetzt, der für Verknüpfungsoperationen doppelt dargestellt wird (Abbildung 33). Daneben können in einer Listenansicht alle Verknüpfungen zu einer korrespondierenden Selektion von Knoten der beiden Bäume angezeigt werden.

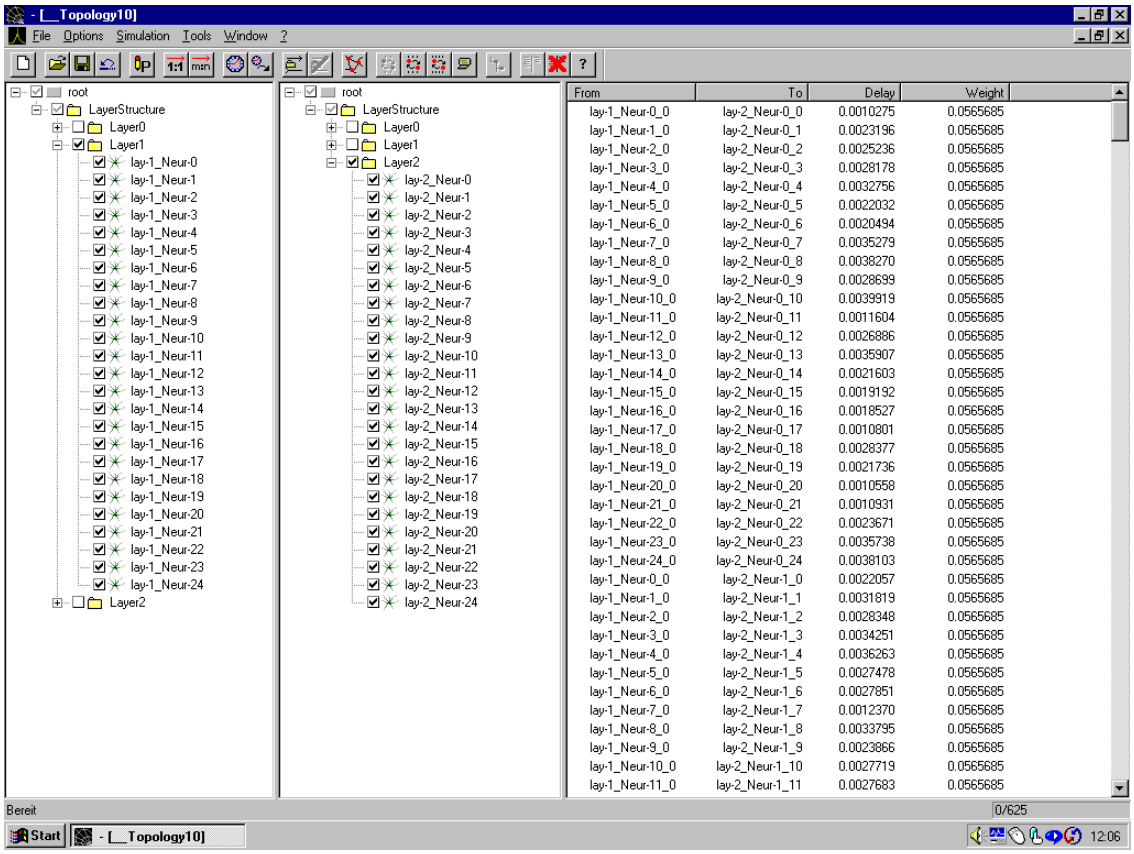


Abbildung 33: Eingabefenster von SPIKELAB. Links sind die beiden Baumstrukturen zu erkennen, welche beide die Hierarchie der aktuell betrachteten Topologie wiedergeben. Rechts ist die Liste der Verbindungen zu sehen, die durch die Selektion der Startneuronen im linken Baum und den Zielneuronen im rechten Baum ausgewählt wurden. (Visualisiert ist ein 3-schichtiges Netzwerk mit je 25 Neuronen pro Schicht, wobei jedes Neuron mit allen Neuronen der benachbarten Schicht verbunden ist.)

Bei der dreidimensionalen Darstellung der Topologien werden die Hierarchieebenen durch die Neuronen umhüllende Quader repräsentiert. Diese Quader können entsprechend den Hierarchieebenen in den Baumansichten expandiert oder kollabiert werden. Abbildung 34a gibt die Sicht auf ein dreischichtiges Netzwerk wieder, wobei das gesamte Netzwerk zu seiner Umhüllenden kollabiert ist. In Abbildung 34b ist das gleiche Netzwerk dargestellt, wobei hier jedoch der das gesamte Netzwerk umhüllende Quader zu den Umhüllenden der einzelnen Neuronenschichten expandiert wurde. Die Verbindungen zwischen den Schichten stehen stellvertretend für alle Verbindungen, die zwischen Neuronen der betreffenden Schichten existieren. In den Abbildung 34c und Abbildung 34d wurde das Netzwerk weiter expandiert. Abbildung 34c zeigt das Netzwerk mit einer expandierten Neuronenschicht, wobei Verbindungen der einzelnen Neuronen zu den nächstgelegenen Kontaktpunkten der benachbarten Umhüllenden verlaufen. Abbildung 34c zeigt schließlich das vollständig expandierte Netzwerk mit allen Neuronen und Verbindungen. Es handelt sich um ein Netzwerk mit 25 Neuronen in jeder Schicht und einer vollständigen Vernetzung benachbarter Schichten, d.h. ein Neuron ist mit allen Neuronen der Nachbarschicht verbunden.

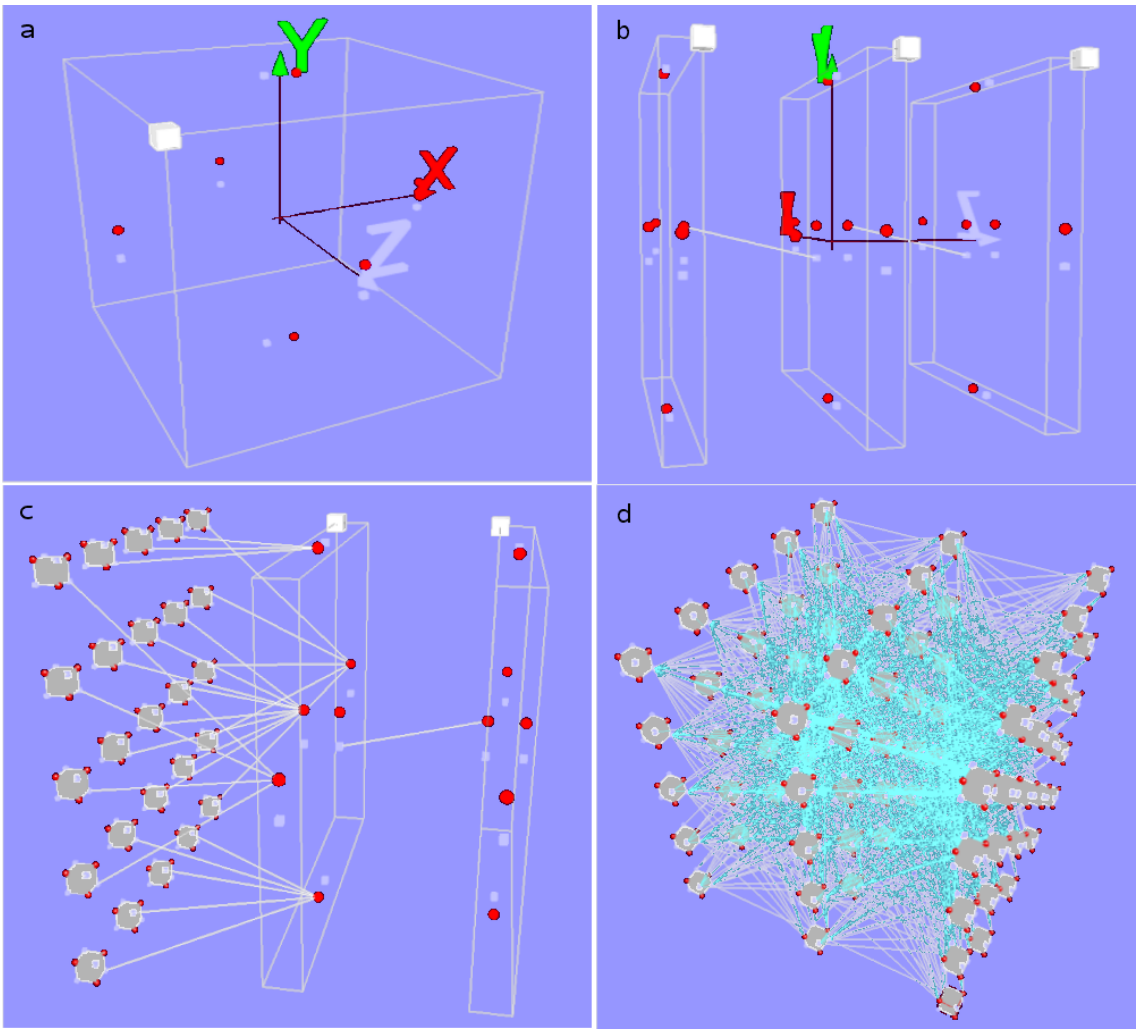


Abbildung 34: 3D-Visualisierung eines dreischichtigen Netzwerks in SPIKELAB. Oben links (a) ist die oberste Hierarchieebene in Form eines das gesamte Netzwerk umhüllenden Quaders zu sehen. Oben rechts (b) zeigt die nächste Hierarchieebene, in der die Umhüllenden der einzelnen Neuronenschichten erkennbar sind. Die Verbindungen zwischen den Schichten geben an, dass grundsätzlich Verbindungen zwischen Neuronen der Schichten existieren. Unten links (c) ist eine der drei Schichten expandiert worden. Hier sind einzelne Neuronen und deren Verbindungen zu den nächstgelegenen Kontaktpunkten der Nachbarschicht sichtbar. Unten rechts (d) sind alle Schichten expandiert und sämtliche Neuronen und deren Verbindungen dargestellt.

Topologien können entweder manuell oder unter Verwendung unterschiedlicher Netzwerkgeneratoren in das Simulationssystem eingegeben werden. Die zurzeit implementierten Netzwerkgeneratoren decken die gängigen Netzwerkstrukturen ab, indem sie die Generierung von voll vernetzten Netzwerken und geschichteten Strukturen zulassen. Durch eine entsprechende Standardisierung der Schnittstelle für derartige Netzwerkgeneratoren wie auch deren Auslagerung in eine Bibliothek, die dynamisch gebunden wird, können diese leicht erweitert werden. Neue Netzwerkgeneratoren werden nach dem erneuten Übersetzen der entsprechenden Bibliothek automatisch vom System erkannt und stehen umgehend zur Verfügung, ohne dass Eingriffe in der Hauptapplikation erforderlich wären. Die notwendigen Schritte zur Einbindung sind in dem folgenden Kapitel zur Implementierung kurz erläutert und im Detail dem Anhang zu entnehmen. Bei einer manuellen Eingabe der Netzwerke werden einfache Vernetzungsoperationen, wie die Eins-zu-Eins-Vernetzung, und die Eins-zu-N-Vernetzung markierter Neuronen unterstützt. Eine rudimentäre Fehlerbehandlung unterbindet dabei nicht durchführbare Vernetzungsoperationen. Die Modifikation von

Parametern der Neuronen und deren Verbindungen können sowohl einzeln als auch im Verbund für mehrere gleichartige Neuronen und Verbindungen durchgeführt werden.

4.1.2. Automatische Partitionierung

Basierend auf Algorithmen zur Graphen-Partitionierung bietet SPIKELAB Methoden, um Topologien für eine verteilte Simulation automatisch aufzuteilen. Als Kostenfunktion gehen bei der automatischen Partitionierung die durchschnittliche Laufzeit zur Berechnung der Neuronenmodelle wie auch Verbindungskosten ein, die anhand der Verzögerung zwischen zwei Neuronen bestimmt werden. Zudem fließen weitere Parameter ein, wie ein Leistungsindex für die einzelnen Zielrechner und die Verbindungskosten für den Nachrichtentransfer zwischen den Rechnern. Die Verbindungskosten werden dabei in Relation zu der Bandbreite der Netzwerkverbindung bestimmt.

Die Implementierung der automatischen Partitionierung existiert für eine Datenbank basierte Repräsentation der Topologie (siehe auch [Schulz1999]). Da die Topologie bei der weiteren Entwicklung von SPIKELAB in eine speicherbasierte Repräsentation überführt wurde, ist für den Einsatz der automatischen Partitionierung mit der aktuellen SPIKELAB Version eine Portierung des Moduls notwendig.

4.1.3. Ereignisgetriebene und verteilte Simulation

Der Kern zur ereignisgetriebenen und verteilten Simulation pulsverarbeitender neuronaler Netze kann sowohl über die graphische Oberfläche gesteuert werden als auch über eine Konsole. Diese Implementierung erlaubt es die Simulation auf beliebigen Plattformen zu starten und zu steuern werden. Ein Subsimulator wird mit Hilfe eines Objekts, der Partition, konfiguriert. Eine Partition kann durch einen Parser aus einer Datei generiert oder unmittelbar innerhalb der graphischen Oberfläche aus den dort eingegeben Daten gewonnen werden. Dadurch sind Simulationen von Netzwerken kleinerer oder mittlerer Größe mit der Einzelplatzversion des Simulators direkt über die Oberfläche möglich, die auch zur Eingabe der Netzwerke dient. Die Initialisierung einer verteilten Simulation ist in der aktuellen Version des Simulators nur über eine Datei möglich.

Die zur Konfiguration des Simulators eingesetzte Partition repräsentiert einen Teil oder das gesamte Netzwerk und beinhaltet alle notwendigen Informationen zur Simulation desselben. Sie umfasst auch Informationen über benachbarte Partitionen in einer verteilten Simulation. Eine Ereignisliste, der Simulatorkern (Klasse *SimulationEngine*)[38] und die logischen Prozesse bilden die Hauptkomponenten eines Subsimulators. Der Simulatorkern entnimmt im Verlauf der Simulation die zeitlich aufsteigend sortierten Ereignisse aus der Ereignisliste und führt sie den logischen Prozessen zur Verarbeitung zu. Diese speisen wiederum neue Ereignisse in die Ereignisliste ein. Die Ereignisse sind entweder Pulse, die zwischen den Neuronen ausgetauscht werden, oder Kontrollereignisse, die dazu dienen, die zeitliche Konsistenz der Simulation zu gewährleisten oder die Simulation zu steuern.

[38] Klassennamen werden an geeigneter Stelle unterstrichen in kursiver Schrift angegeben.

4.1.4. Analyse von Simulationsergebnissen

Bei einer Simulation können sowohl die Ereignisse aufgezeichnet werden, welche von den logischen Prozessen generiert werden, als auch die Veränderungen von Modellvariablen.

Im Rahmen des Projektes ist ein Tool entstanden, welches erlaubt, insbesondere die Ereignisse darzustellen und einfache statistische Analysen darauf auszuführen. Das Tool ist für die Datenbank basierte Variante von SPIKELAB implementiert worden und gestattet es Gruppen, so genannte Populationen, für die Beobachtung und Analyse der Ereignisse zu definieren.

Darüber hinaus können die Daten in jedem anderen Programm leicht weiter verarbeitet werden, da die Daten in ASCII-Format in der Form Zeitpunkt, Prozessidentifikation, Werttyp und Wert abgelegt werden. So bietet sich als Ergänzung z. B. die Einbindung kommerzieller Mathematikprogramme wie Matlab an, da dort schon viele Verfahren zur Auswertung solcher Daten bereitgestellt werden.

4.1.5. RACER – Hardwarebeschleunigung

Die ereignisgetriebene Simulation pulsverarbeitender neuronaler Netze nutzt bereits grundlegende Eigenschaften dieser Netze für eine beschleunigte Simulation aus. Um Echtzeitbedingungen zu erreichen ist jedoch der Einsatz von Spezialhardware unverzichtbar. Zudem können geeignete Hardwaremodule die Verarbeitung kontinuierlicher Signale übernehmen, welche in der ereignisgetriebenen Simulation schlecht abzubilden sind. In SPIKELAB ist die RACER-Hardware für diesen Zweck vorgesehen. Es handelt sich dabei um eine Erweiterungskarte für den PCI-Bus. Sie kann bis zu drei Module aufnehmen, welche entweder bis zu zwei programmierbare Bausteine tragen können oder beliebige Schaltkreise, welche eingangsseitig über eine digitale 32 Bit Schnittstelle ansprechbar sind und ausgangsseitig ihre Ergebnisdaten in Form von 32 Bit Worten liefern. Die Karte versorgt die drei Module mit Daten und transferiert die von den Modulen erzeugten Daten in den Speicher des Rechners. Neben einer Schnittstelle für die Programmierung von programmierbaren Bausteinen auf den Modulen existieren keine weiteren Steuerleitungen. Es handelt sich daher um eine Datenflussarchitektur, die sowohl für eine maximale Datentransferrate als auch für eine geringe Latenz ausgelegt ist.

4.2. Softwarearchitektur

Das Datenmodell, welches über die graphische Oberfläche erstellt und manipuliert wird, wird durch die Topologie (*CTopology*) repräsentiert. Diese Klasse beinhaltet alle Informationen über die an der Simulation beteiligten Neuronen, deren Parameter und ihre Aufteilung in Partitionen (*Partition*). Die Topologie repräsentiert also das zu simulierende neuronale System, welches ein Netzwerk aus mehreren komplexen oder einfachen Knoten (*Node*) ist. Komplexe Knoten repräsentieren wiederum Netzwerke (*Network*) von Knoten, so dass sich eine hierarchische Anordnung der Knoten in Form eines Baumes ergibt. Daher ist der Wurzelknoten einer Topologie ein Netzwerk (*Network*). Entsprechend wurde in der graphischen Oberfläche ein Baum als Repräsentation gewählt. Neuronen (*Neuron*) bilden die Endknoten in dieser Hierarchie bzw. die Blätter des Baums.

Simuliert werden letztlich die Partitionen einer Topologie. Dabei speisen die Modelle Ereignisse in eine Ereignisliste (*EventList*) ein. Das früheste darin enthaltene Ereignis

wird jeweils dem entsprechenden Modell zur Verarbeitung übergeben. Der Ablauf der Simulation wird von einer Klasse gesteuert, die eine Ableitungen der Klasse *Simulator* ist.

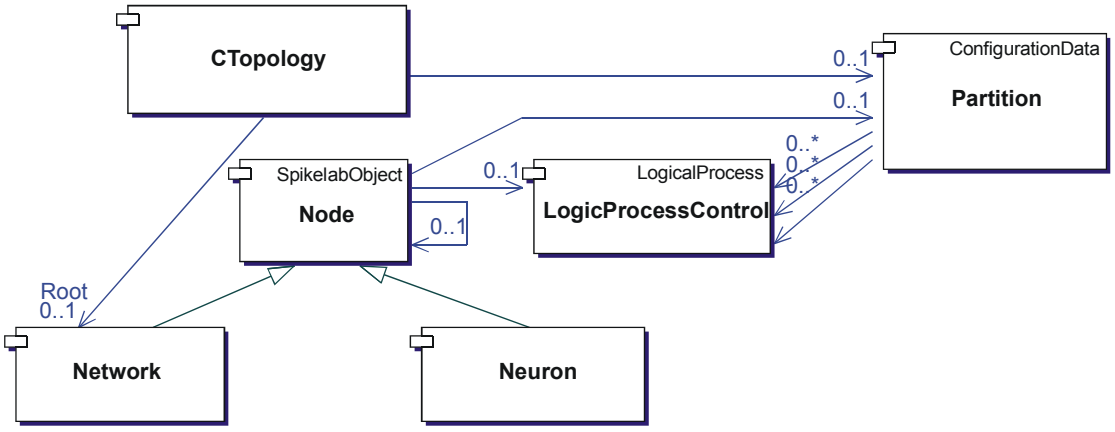


Abbildung 35: Eine Topologie bestehend aus Knoten, die Netzwerke oder Neuronen repräsentieren. Die simulierbare Repräsentation eines Knotens ist der logische Prozess. Die logischen Prozesse werden in Partitionen zusammengefasst[39].

Die Ableitungen der Klasse *Simulator* sind die Klassen *SimulatorEngine* und *DistributedSimulationCoordinator*. *SimulatorEngine* besitzt wiederum die Ableitungen *StandAloneSimulator* und *SubSimulator*. *StandAloneSimulator* implementiert eine Einzelplatzversion des Simulators. *SubSimulator* ist hingegen ein Teil eines verteilten Simulators, welcher über den *DistributedSimulationCoordinator* gesteuert wird.

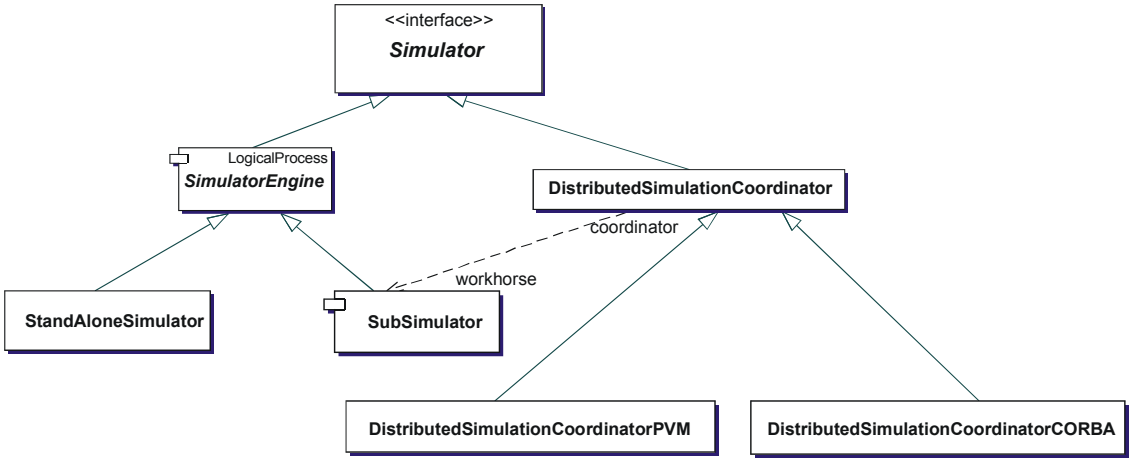


Abbildung 36: Vererbungshierarchie der unterschiedlichen Simulatorklassen.

Beide Varianten können jedoch in gleicher Weise als Simulator aus der graphischen Oberfläche heraus angesprochen werden, so dass weitgehend transparent ist, ob die Simulation verteilt abläuft oder nicht. Ausgenommen hiervon ist natürlich die Initialisierung, da bei der verteilten Simulation eine entsprechende Zuordnung von Partitionen zu Rechnern erfolgen muss. Da dem *StandAloneSimulator* gegenüber dem *SubSimulator* die Klassen zur Kommunikation fehlen, der Kern des Simulators (vergl.

[39] Alle Diagramme, welche die Klassenhierarchien wiedergeben, erwenden die UML Notation [FowSco1998],[LeeTep1997]

SimulatorEngine) jedoch nahezu identisch ist, wird im Folgenden der *SubSimulator* detaillierter dargestellt.

4.2.1. Die Simulatorschnittstelle

Die Simulatorschnittstelle definiert im Wesentlichen folgende Methoden:

```
configure(ConfigurationData*)
run(const Time)
pause()
cont()
stop()
restart()
reset()
insertEvents(const vector<Event*> events)
const list<SimulatorState> state()
const list<SimulatorTime> time()
const list<SimulatorProfile> profile()
```

Mit `configure` wird ein Simulator initialisiert, wobei beim Aufruf der Methode ein Parameter der Klasse `ConfigurationData` erwartet wird. `ConfigurationData` repräsentiert als Basisklasse alle im System vorgesehenen Konfigurationsdaten, zum Beispiel die Klasse Partition, ein *Filestream* oder einfach einen Dateinamen. Mit `run` wird die Simulation gestartet. Der Parameter der Methode gibt an, für welche virtuelle Zeitdauer simuliert werden soll. Ein gestarteter Simulationslauf kann mit `pause` vorübergehend unterbrochen, mit `cont` fortgesetzt oder mit `stop` angehalten werden. Die Methoden `restart` und `reset` setzen den Simulator zurück, wobei `restart` das geladene Projekt auf die Simulationszeit Null zurücksetzt und `reset` den gesamten Simulator zurücksetzt sowie ein eventuell geladenes Projekt entfernt. Die Methode `insertEvents` ermöglicht es, zusätzliche Ereignisse in die Simulation einzubringen. Die drei Methoden `state`, `time` und `profile` dienen dazu, den aktuellen Zustand, die aktuelle Simulations- und Laufzeit und eine Leistungsstatistik abzufragen. Die Rückgabewerte sind Listen, da die Methoden unter Umständen für einen verteilten Simulator aufgerufen werden und in einem solchen Falle für jeden Subsimulator die entsprechenden Informationen zurückgeliefert werden. Die Kapselung der Rückgabewerte in entsprechenden Klassen `SimulatorState`, `SimulatorTime` und `SimulatorProfile` ermöglicht eine nachträgliche Ergänzung dieser Klassen, ohne dass bereits bestehende Auswertefunktionen geändert werden müssen.

4.2.2. Subsimulator

Gegenüber der Einzelplatzversion des Simulators besitzen die Subsimulatoren des verteilten Simulators eine Kommunikationsschnittstelle, welche aus einem *Transmitter* und einem *Receiver* besteht. Diese beiden Klassen legen die Schnittstelle fest, über die der Subsimulator auf das Netzwerk zugreift, um mit anderen Sub-Simuatoren oder der Simulationskontrolle zu kommunizieren. Entscheidend ist, dass die Subsimulatoren lediglich eine lokale Synchronisation durchführen, also mit den unmittelbaren Nachbarn, und nicht über die Simulationskontrolle synchronisiert werden. Hierfür muss der Subsimu-

lator seine unmittelbaren Nachbarn kennen, aus deren Simulationsfortschritt der sichere Horizont für die lokale Simulation gebildet wird. Da entsprechende Informationen über die nächsten Nachbarn und Referenzen auf sie von dem verwendeten Nachrichtensystem bereitgestellt werden, sind für die Implementierung mit unterschiedlichen Nachrichtensystemen Ableitungen von *Receiver*, *Transmitter* und *SubSimulator* vorgesehen. Die Basisklasse *Simulator* stellt eine einheitliche Schnittstelle für alle Ableitungen sicher.

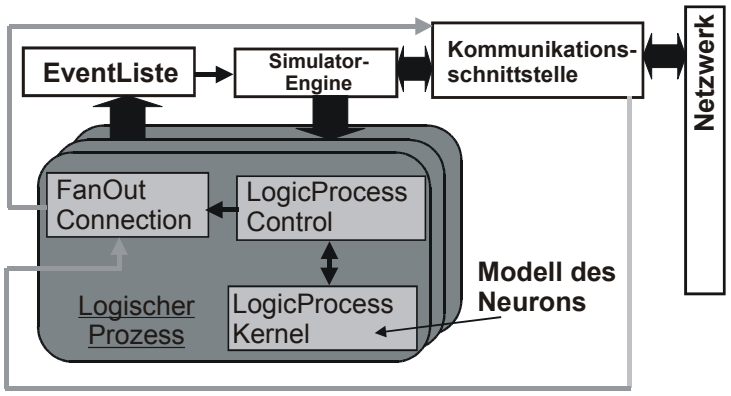


Abbildung 37: Subsimulator

Abbildung 37 ist zu entnehmen, dass ein *SubSimulator* aus einer Menge logischer Prozesse besteht. Ein logischer Prozess besteht wiederum aus den drei Klassen *LogicProcessControl*, *LogicProcessKernel* und *FanOutConnection*.

4.2.2.1. Logischer Prozess

Die Klasse *LogicProcessControl* stellt die äußere Schale eines Neuronenmodells dar, welche den *LogicProcessKernel* umschließt und letztlich dazu dient, die Schnittstelle des Modells möglichst einfach zu gestalten und verwaltungstechnische Aufgaben, die im Zuge des Simulationsablaufes anfallen, von diesem fern zu halten. Deshalb werden auch die vom Modell erzeugten Ereignisse durch diese Klasse aufgenommen und an die Klasse *FanOutConnection* weitergeleitet, welche die Verteilung der Ereignisse vornimmt. Hierfür werden Kopien des Ereignisses mit den Adressen und den individuellen Verzögerungen der Nachfolger versehen und in die Ereignisliste eingetragen.

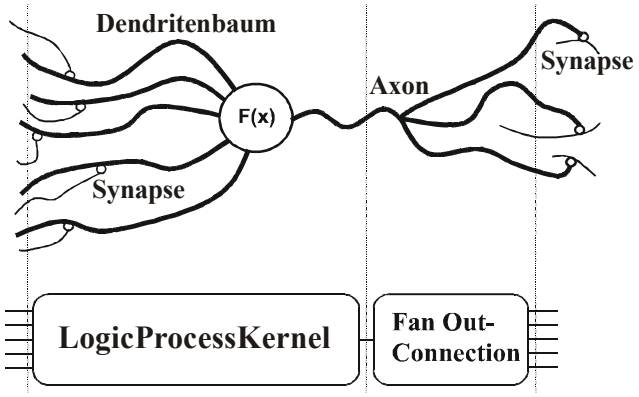


Abbildung 38: Modellierung eines Neurons durch *LogicProcessKernel* und *FanOutConnection*

Da die Verzögerungen zu den Nachfolgern bekannt sind, werden die Ereignisse wahlweise in zeitlich aufsteigender oder absteigend sortierter Reihenfolge in die

Ereignisliste eingefügt. Dies kann gegebenenfalls bei der Implementierung der Ereignisliste für Optimierungen genutzt werden. Abbildung 38 zeigt schematisch die Analogie zwischen einem biologischen Neuron und dessen Abbildung in den entsprechenden Klassen des Simulators.

An die Klasse *LogicProcessKernel* werden lediglich zwei wesentliche Anforderungen gestellt:

1. Die Modellimplementierung darf die zeitliche Konsistenz der Simulation nicht verletzen. Das bedeutet, dass aufgrund von eingehenden Ereignissen nur Ereignisse mit gleichem oder größerem Zeitstempel generiert werden dürfen.
2. Die Modellimplementierung muss bei der Verarbeitung eines Eingangspulses sicherstellen, dass eine potentielle Aktivität in der Zukunft auf Basis der aktuellen Eingabesituation bestimmt wird. Wird aufgrund einer Verarbeitung kein Ereignis zurückgeliefert, so ist dies gleichbedeutend mit der Aussage, dass der Prozess ohne weitere Eingaben inaktiv bleibt.

Zusätzlich sollten von dem Modell Refraktärphasen in Form von entsprechenden *look aheads* angegeben werden, damit eine effiziente Simulation möglich ist.

Um einerseits die gemeinsame Nutzung von Parametern zu ermöglichen, andererseits den Zugriff auf die Parameter zu formalisieren, wurden die Klassen *ParameterManager* und *ParameterSet* eingeführt, über welche die Manipulationen und der Austausch der Parameter stattfinden. Für die graphische Oberfläche existiert ein entsprechendes Pendant, der *ParameterManagerGUI*, der mit Hilfe des *ParameterManager* implementiert ist. Eine zusätzliche Klasse für die graphische Repräsentation ist eingeführt worden, damit die Parameterverwaltung plattformunabhängig, sowohl mit, als auch ohne eine graphische Oberfläche zur Verfügung steht. Das *ParameterSet* ist ein Container für alle Parameter eines Modells, wobei darin auch Verweise auf gemeinsam genutzte Parameter enthalten sein können. Die gemeinsam genutzten Parameter werden durch den *ParameterManager* angelegt und verwaltet. Durch den Containeransatz ist eine Trennung zwischen Parametereingabe und –manipulation einerseits und der Simulation andererseits gewährleistet.

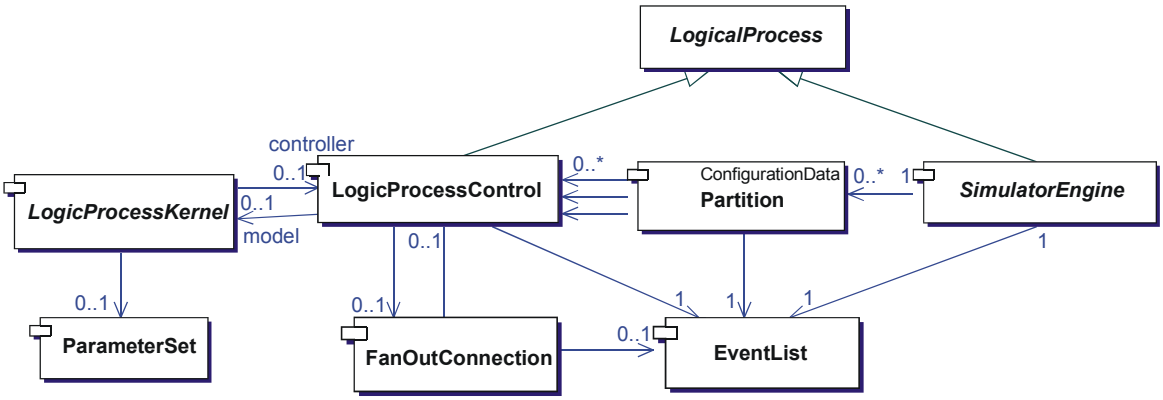


Abbildung 39: Vererbungshierarchie des logischen Prozesses

Für die Implementierung eines neuen Neuronenmodells sind daher Ableitungen der Klassen *LogicProcessKernel* und *ParameterManager* zu implementieren. Für die graphische Unterstützung muss zudem eine Ableitung von *ParameterManagerGUI* implementiert werden.

Die Zahl der logischen Prozesse ist einerseits durch den verfügbaren Speicher beschränkt und andererseits durch die Rechenleistung des zur Simulation verwendeten Computersystems. Abhängig von der Auslastung des Einzelplatzsystems bietet sich die Verteilung der Simulation auf mehrere Rechner an. Dies ist spätestens dann sinnvoll, wenn der Simulator als Einzelplatzsystem bei der Bearbeitung Daten auf die Festplatte auslagern muss. Der Speicherbedarf wird dabei nicht nur durch die Größe des zu simulierenden Netzwerks bestimmt, sondern auch durch die Aktivität im System, da entsprechend viele Ereignisse in der Ereignisliste verwaltet werden müssen.

4.2.2.2. Zentrale Simulationsschleife

Die *SimulatorEngine* entnimmt in der Grundbetriebsart der Ereignisliste das früheste Ereignis und leitet es an den im Ereignis angegebenen logischen Prozess weiter. Wie der Abbildung 39 zu entnehmen ist, besitzen die *SimulatorEngine* und die *LogicProcessControl* eine gemeinsame Basisklasse - *LogicalProcess*. Dies trägt der Tatsache Rechnung, dass die *SimulatorEngine* genauso wie der logische Prozess als Senke für Ereignisse fungiert. So kann die *SimulatorEngine* in gleicher Art als Zieladresse in ein Ereignis eingetragen werden und vor allem während der Simulation in gleicher Weise behandelt werden.

Die *LogicProcessControl* wertet die im Ereignis übermittelten *look aheads* aus, um einen sicheren Horizont zu bestimmen. Innerhalb dieses Horizonts können Ereignisse versendet werden, die von dem durch die *LogicProcessControl* verwalteten *LogicProcessKernel* generiert werden. Wird von dem *LogicProcessKernel* ein Ereignis mit einem Zeitstempel generiert, der nicht mehr innerhalb des sicheren Horizonts liegt, so wird das Ereignis durch die *LogicProcessControl* gespeichert und ein so genanntes *wake-up* Ereignis in die Ereignisliste eingetragen. Dieses Ereignis trägt den Zeitstempel des von dem *LogicProcessKernel* generierten Ereignisses und dient dazu, die *LogicProcessControl* wieder aufzurufen, wenn bis zum Eintreten des berechneten Ereignisses keine weiteren Berechnungen durch den *LogicProcessKernel* stattgefunden haben. Eine erneute Berechnung ist jedoch notwendig, wenn zwischen der letzten Berechnung und dem Eintreffen des *wake-up* Ereignisses ein weiteres Ereignis eintrifft. In einem solchen Falle verfällt das zuvor generierte *wake-up* Ereignis.

Zusätzlich kann in der *SimulatorEngine* ein Modus aktiviert werden, in dem der Ereignisliste neben dem frühesten Ereignis noch alle weiteren Ereignisse entnommen werden können, deren Bearbeitung unabhängig voneinander geschehen kann. In dieser Betriebsart werden Ereignisse aus der Liste entnommen, solange deren Zeitstempel innerhalb eines sicheren Bereichs liegen, der durch den Zeitstempel des zuerst entnommenen Ereignisses und das Minimum aus allen Verzögerungen zu dessen Nachfolgern bestimmt wird[40]. Der Bereich wird weiter eingeschränkt, sofern die Summe aus Zeitstempel und minimaler Verzögerung von einem der im Folgenden entnommenen Ereignisse geringer ist als die aktuelle Bereichsgrenze. Ob ein Ereignis aus der Liste in den sicheren Bereich fällt, wird innerhalb der zentralen Simulationsschleife mit Hilfe der Methode *checkIfNextIsSecure()* überprüft, welche auch die Anpassung des Bereichs vornimmt. Durch diese Kapselung können auch andere Verfahren für die Verarbeitung mehrerer Ereignisse leicht integriert werden. Die gewählte Implementierung stellt einen Kompromiss zwischen den overheadbedingten Einbußen bei der Simulationsleistung

[40] Das Minimum aus den Verzögerungen zu allen Nachfolgern wird während der Initialisierung bestimmt und muss nicht bei jedem Aufruf dieser Methode erneut ermittelt werden.

und der maximal erreichbaren Zahl gemeinsam verarbeitbarer Ereignisse dar und folgt der im Abschnitt 3.3.1 angeführten Optimierungsstrategie.

4.2.3. Subsimulator in einer verteilten Simulation

Bei einer verteilten Simulation reicht die Synchronisation über die Ereignisliste des Simulators nicht mehr aus. Um die zeitliche Konsistenz in jedem an der Simulation beteiligten Subsimulator zu gewährleisten, müssen sich die Subsimulatoren untereinander synchronisieren oder zentral über einen Kontroller synchronisiert werden. Eine zentrale Synchronisation verbietet sich jedoch aufgrund des dafür notwendigen Kommunikationsaufwands. Für eine dezentrale Synchronisation einer ereignisgetriebenen Simulation haben sich die zwei in Abschnitt 3.1.2.2 beschriebenen Verfahren etabliert, die konservative und die optimistische Synchronisation.

Die Simulation in SPIKELAB wird konservativ synchronisiert. Bei der konservativen Synchronisation darf ein Subsimulator des verteilten Simulators nur sichere Ereignisse verarbeiten. Ein Ereignis ist dann als sicher zu bezeichnen, wenn zweifelsfrei feststeht, dass von benachbarten Subsimulatoren, die Verbindungen zu dem betrachteten Subsimulator besitzen, keine Ereignisse eintreffen können, die früher als das zu verarbeitende Ereignis stattfinden.

4.2.4. Koordination der verteilten Simulation

Aus dem vorangehenden Abschnitt geht hervor, dass die Subsimulatoren eine dezentrale konservative Synchronisation implementieren. Jedoch muss der Fortschritt der Simulation an zentraler Stelle nachvollziehbar sein. Ebenso muss die verteilte Simulation von zentraler Stelle aus initialisiert, gestartet und gestoppt werden können. Diese Aufgaben übernimmt die Klasse *DistributedSimulationCoordinator* (vergleiche Abbildung 36). Die Implementierung der Klasse ist in der Regel eng mit dem verwendeten Nachrichtensystem gekoppelt, da die Subsimulatoren auf entfernten Rechnern gestartet, die Konfigurationsdaten an die Subsimulatoren verteilt und die Ergebnisse der Simulation wieder zurücktransferiert werden müssen. Diese Abhängigkeit liegt insbesondere in speziellen, parallelen Rechnersystemen vor, da hier unter Umständen Transferprotokolle, die typischerweise in Rechnernetzwerken zur Verfügung stehen, nicht vorhanden sind.

Der Koordinator führt eine Liste der für die Simulation zur Verfügung stehenden Rechner und ist in der Lage, eine Datei zu verarbeiten, in der die gewünschte Verteilung der Konfigurationsdaten festgehalten ist (Dies ist die so genannte Hostdatei mit der Endung *.hst). Die derzeitige Implementierung bietet zudem die Möglichkeit, die Simulation zufällig auf die verfügbaren Rechner zu verteilen. In zukünftigen Implementierungen könnten die Konfigurationsdaten zusätzlich bezüglich ihres Leistungsbedarfs eingestuft werden, so dass eine entsprechende Zuordnung zu den bereits nach Leistungsfähigkeit klassifizierbaren Rechnern erfolgen kann.

Über die Initialisierung hinaus stellt der *DistributedSimulationCoordinator* das Interface zur Steuerung der Simulation bereit. Dadurch lässt sich der verteilte Simulator genauso wie die Einzelplatzversion bedienen.

4.2.5. Behandlung "zufälliger" Ereignisse

Um Prozesse im Simulator zu integrieren, welche Ereignisse selbständig, in regelmäßiger oder zufälliger Folge, generieren, wird im Simulator zwischen normalen Prozessen

und Generatorprozessen unterschieden. Generatorprozesse erhalten zu Beginn der Simulation ein so genanntes Generator-*wake-up*-Ereignis. Der Generatorprozess muss daraufhin die in der Zukunft stattfindenden Ereignisse bestimmen und sie in die Ereignisliste eintragen. Jedoch sollte die Zahl der generierten Ereignisse so bemessen sein, dass die Ereignisliste nicht zu sehr gefüllt, aber eine ausreichend weite Vorausschau erreicht wird, damit der Generatorprozess nicht unnötig oft aufgerufen werden muss. Dies abzuwägen liegt in der Verantwortung des Modellschreibers, wobei im Zweifelsfalle die Zahl der zu generierenden Ereignisse über einen Modellparameter steuerbar gemacht werden kann. Mit diesem Parameter kann dann in Abhängigkeit einer konkreten Simulation individuell Einfluss auf die Zahl der generierten Ereignisse genommen werden. Der Modellschreiber muss zudem selber darauf achten, dass sein Generatorprozess wieder rechtzeitig aufgerufen wird, indem er ein entsprechendes Generator-*wake-up*-Ereignis in die Ereignisliste einträgt, da der Simulator lediglich das erste *wake-up* Ereignis zu Beginn der Simulation bereitstellt. In der vorliegenden Implementierung stehen drei Generatoren zur Verfügung: Die Klasse *SpikeTrain*, welche in einem einstellbaren Intervall Pulse erzeugt, die Klasse *RandomSpikeTrain*, mit der zufallsgesteuert mit einer einstellbaren Wahrscheinlichkeitsverteilung Pulse generiert werden, und die Klasse *SpikeBitmap*, welche binär codierte Bilddateien einliest und die gesetzten Bildpunkte nach einem einstellbaren Intervall in die Simulation einspeist.

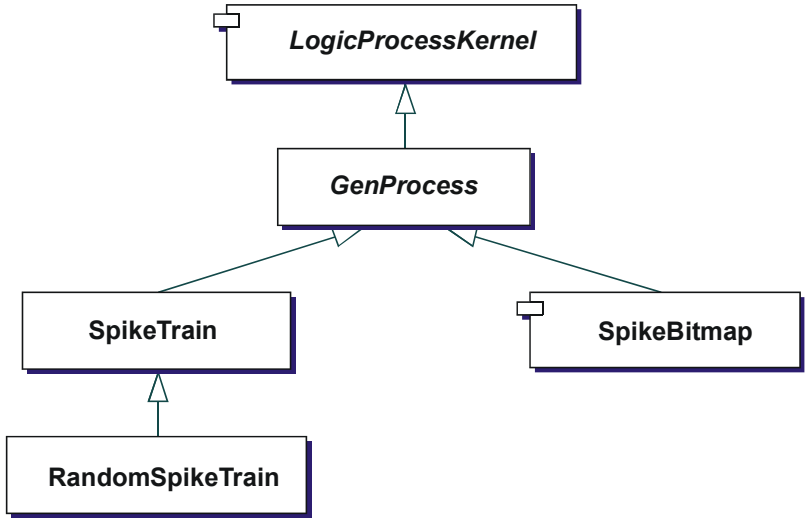


Abbildung 40: Generatorprozesse in Spikelab

4.2.6. Realisierung von Lernvorgängen

Lernverfahren sind in der vorliegenden Implementierung von Spikelab noch nicht vorhanden. Für eine Umsetzung bieten sich Lernregeln an, die zum Neuron lokal sind, da durch die ereignisgetriebene Simulation immer zum Neuron lokale Berechnungen durchgeführt werden und die Berücksichtigung globaler Zusammenhänge, wie zum Beispiel der Gesamtfehler über die Ausgaben mehrerer Neuronen, die gleichzeitige Auswertung mehrerer Neuronen, unabhängig von ihrer Aktivität bedingt. Eine solche Herangehensweise steht im Widerspruch zur ereignisgetriebenen Simulation und kommt eher einer Zeitscheibensimulation entgegen, da in einer solchen Simulation ohnehin in jedem Zeitschritt der Zustand aller Neuronen aktualisiert wird.

Langsam veränderliche Parameter, wie zum Beispiel Adrenalinpegel oder hormonelle Einflussgrößen, welche mehrere Neuronen gleichermaßen beeinflussen, können auch in

der ereignisgetriebenen Simulation noch relativ effizient simuliert werden. Entscheidend ist in diesem Zusammenhang die Geschwindigkeit, mit der sich solche Parameter verändern, denn jede wesentliche Veränderung muss zu einem Ereignis führen, welches an alle Neuronen gesendet wird, die auf den Parameter zugreifen.

Da lokal wirkende Lernregeln auch aus neurophysiologischer Sicht [SenTsoMar1997] plausibel erscheinen, hingegen Verfahren, die zum Beispiel auf globale Inhibition durch ein Neuron angewiesen sind, eher unwahrscheinlich sind, bietet Spikelab eine gut geeignete Plattform für die Entwicklung solcher Lernverfahren. Über die Anpassung der synaptischen Gewichtung hinaus sind auch Lernverfahren denkbar, welche die Verzögerungen der Verbindungen anpassen. Durch die explizite Berücksichtigung dieser Systemeigenschaft in Spikelab lassen sich demnach leicht auch Lernverfahren implementieren, welche die Verzögerungen anpassen.

4.2.7. Hardwareanbindung

Zur Evaluierung von Hardware-Implementierungen wurden entsprechende Klassen in den Simulator integriert. Von diesen Klassen können Modulobjekte instanziiert werden. Die Modulobjekte entsprechen den Modulen auf der RACER-Karte und werden von der Anwenderschicht des Treibers bereitgestellt (siehe 6.1.3.2.2 Anwenderbibliothek). Es wurden Klassen für die Einbindung einer in Hardware realisierten Ereignisliste und zur Einbindung von in Hardware realisierten Neuronenmodellen integriert. Diese Klassen sorgen in erster Linie für eine entsprechende Aufbereitung der Daten aus dem Simulator, damit sie von der Hardware verarbeitet bzw. nach ihrer Bearbeitung wieder in den Simulationsablauf eingespeist werden können. Es wird dabei stets das Ziel verfolgt, möglichst viele Daten mit wenigen Übertragungen zu transferieren und den Aufwand für die Aufbereitung der Daten so gering wie möglich zu halten, da für kleine Datenmengen die Leistung der Hardware alleine durch die Datenübertragung und nicht durch die Modul-Implementierung bestimmt wird.

Bei der Ereignisliste wird das Objekt *evlRep* der Klasse *Eventlist* mit einer Instanz der entsprechenden Klasse zur Hardwareeinbindung initialisiert. Es stehen hierfür die Klassen *RacerQueue*, *RacerQueueRingbuffer*, sowie *RacerQueueFS* und *RacerQueueFSRingbuffer* zur Verfügung. Die FS-Varianten dienen zur Einbindung der Implementierung, die auf dem Fishspear-Algorithmus basiert, wohingegen die anderen Varianten zur Einbindung der Implementierung dienen, die auf dem Calendarque-Algorithmus beruht. Die Ringbuffervarianten verwenden eine Zwischenpufferung in der Anwenderschicht des Treibers und erhöhen so die Performance für kleinere Datenpakete, da diese potentiell durch dieses Verfahren zu größeren Paketen zusammengefasst werden.

Für die Einbindung der Hardware-Neuronenmodelle liegen zwei unterschiedliche Implementierungen vor. Zum einen eine Ableitung der Klasse *SpikingNeuron*, die Klasse *RacerIAF*, und zum anderen eine Erweiterung der Klasse *LogicProcessControl*. Wird die Klasse *RacerIAF* verwendet, so wird für jedes Neuron, das durch die Hardware berechnet wird, ein Objekt dieser Klasse instanziert, da durch die Klasse *RacerIAF* ein bestimmter *LogicProcessKernel*, nämlich ein in Hardware implementierter, repräsentiert wird. Dieser kann zur gebündelten Übertragung von Daten zur Hardware unmittelbaren Nutzen aus der überladenen Methode *evaluate(const vector<Event*>&, OutSpikeList&)* ziehen, welche nicht ein einzelnes Ereignis, sondern einen Vektor von Ereignissen zur Berechnung im Neuronenmodell bereitstellt.

Die Übertragung der Daten lässt sich noch weiter bündeln, indem alle in Hardware realisierten Neuronenmodelle zentral verwaltet werden. Dies wird durch die Erweiterung der Klasse *LogicProcessControl* erreicht, indem das Modul in dieser Klasse als statische Variable angelegt wird. Außerdem erfolgt die Verarbeitung der Ereignisse durch eine zur Klasse statischen Methode. So können nicht nur die Ereignisse gebündelt werden, die zu einem Neuron übertragen werden, sondern alle Ereignisse, die in einem Verarbeitungszyklus für die in Hardware implementierten Modelle der Ereignisliste entnommen wurden.

4.3. Leistungsbewertung der Software

4.3.1. Vergleich mit einem Zeitscheibensimulator

Für den Vergleich der ereignisgetriebenen Simulation in SPIKELAB mit einem Zeitscheibensimulator wurde der Simulator GENESIS zum Vergleich herangezogen. GENESIS dient häufig zur Simulation biologienaher neuronaler Netze. Er bietet ein sehr umfangreiches Spektrum an Modellierungsmöglichkeiten und zur Skriptsteuerung. Obwohl GENESIS für die Simulation von Abschnitts-Modellen ausgelegt ist, kann der Simulator durch andere Modelle ergänzt werden. Dies geschieht durch Erweiterungen mit entsprechenden C-Programmen, die der GENESIS-API genügen müssen. Die Erweiterungen werden mit einem Compiler übersetzt und werden nicht interpretiert. So wird sichergestellt, dass die neuen Modelle mit der gleichen Performance simuliert werden wie Modelle, die bereits zum GENESIS-Kern gehören. Entscheidend ist, dass bei diesem Vergleich nicht die gesamten Simulationsumgebungen verglichen werden, welche auch die Komponenten für die Erstellung der Netzwerke und die Auswertung der Simulationsergebnisse umfassen, sondern lediglich die Simulationsverfahren. GENESIS wurde hierfür als Referenz eines Zeitscheibensimulators verwendet.

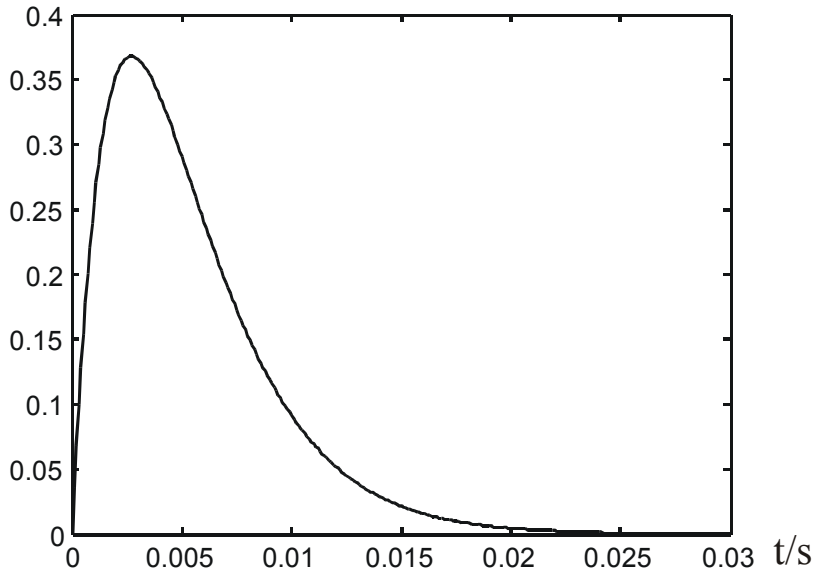


Abbildung 41: Verlauf des postsynaptischen Potentials im Modellneuron

4.3.1.1. Neuronenmodell

Für den Vergleich wurde ein einfaches SRM-Neuron implementiert. Die Zeitkonstanten des Modells wurden in Anlehnung an die Zeitkonstanten kortikaler Neuronen gewählt,

um auch hinsichtlich der Genauigkeit ein möglichst realistisches Bild zu zeichnen. Das postsynaptische Potential folgt der Formel

$$\varepsilon(t) = \frac{t}{\tau} \cdot e^{\frac{-t}{\tau}} \quad mit\ \tau = 2,7ms\,.$$

Ein Schwellenwert von 0,34 stellt sicher, dass bei einem Eingangsgewicht von eins sicher ein Ausgangspuls erzeugt wird.

4.3.1.2. Testnetzwerk

Die Modellneuronen wurden zu vollvernetzten Schichten zusammengefasst. Jede Verbindung erhielt eine zufällige Verzögerung zwischen einer und drei Millisekunden.

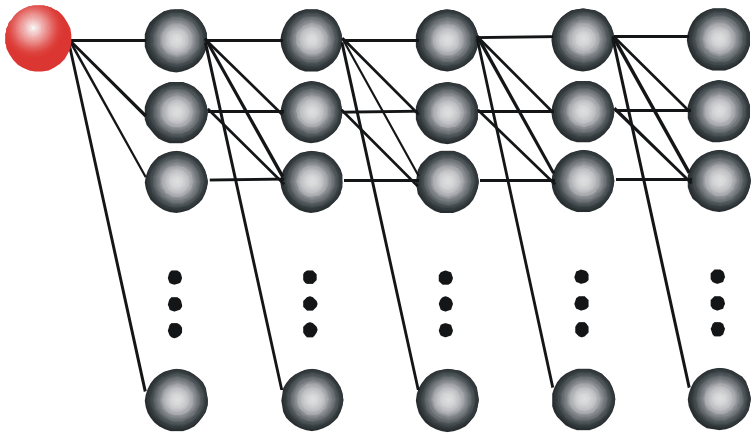


Abbildung 42: Fünfschichtiges, vollvernetztes Testnetzwerk mit *SpikeTrain* als speisendem Prozess

Die Schichtgröße wurde zwischen 10 und 100 Neuronen in 10‘er Schritten variiert. Die Gewichte der Synapsen wurden auf den Kehrwert der Neuronenanzahl der Vorgängerschicht initialisiert, so dass unabhängig von der Neuronenzahl immer die gleiche Aktivierung sichergestellt werden kann. Die Aktivität im Netzwerk kann daher alleine durch das Intervall des speisenden Neurons (*SpikeTrain*) gesteuert werden.

4.3.1.3. GENESIS-Implementierungen

Für GENESIS wurden zwei Implementierungen erstellt. In beiden Fällen handelt es sich um eine Erweiterung des Simulators, die kompiliert und zum Simulatorkern gebunden wird.

Bei der ersten Implementierung wird zu jedem Zeitschritt der Zustand eines jeden Neurons aktualisiert, indem für jeden Eingang eines Neurons der aktuelle Wert des postsynaptischen Potentials abgefragt und mit dem entsprechenden Gewicht der Synapse multipliziert wird. Die zentrale Schleife der Simulation sieht folgendermaßen aus:

```
for(spike = Synapse[i].LastSpike; spike != NULL; spike = spike->prev)
   {channel->Gk += (Synapse[i].weight*psp(t-spike->time,channel));}
   mit
   double psp(double t, register struct tabsynchan_type *channel)
   { return channel->a2*(t/channel->a1)*exp(-t/channel->a1);}
```

Überschreitet das Summenpotential Gk den Schwellenwert, so wird ein Ausgangspuls ausgelöst. Für diese Implementierungsvariante lohnt es sich auch nicht, die in der Funktion *psp()* durchgeführte Berechnung durch eine zuvor berechnete Look-up-Tabelle zu ersetzen, da durch diese Maßnahme keine Beschleunigung der Simulation erreicht wird. Hingegen entstehen durch die Verwendung einer Look-up-Tabelle weitere Rundungsfehler, die zu einer ungenaueren Simulation führen (→ vgl. Abschnitt 3.2.2).

Die zweite Implementierungsvariante verwendet für die Modellierung der Neuronen den Ansatz, der auch in SPIKELAB eingesetzt wird (→ vgl. Abschnitt 3.3.2). Durch diese Art der Modellierung werden lediglich zwei Variablen angepasst, wenn ein Puls an dem betreffenden Neuron eintrifft. Zudem muss bei der GENESIS-Simulation in jedem Zeitschritt geprüft werden, ob der Schwellenwert überschritten wurde. Die Variablen werden jedoch nur angepasst, wenn ein Puls an dem Neuron eingetroffen ist oder der Schwellenwert überschritten wurde. In allen anderen Fällen beschränkt sich der Aufruf darauf, zu jeder Zeitscheibe das Summenpotential aufgrund der beiden Variablen zu bestimmen und zu testen, ob der Schwellenwert überschritten wurde.

Die zweite Implementierungsvariante verhindert in erster Linie, dass zur Bestimmung des Summenpotentials über alle Synapsen des Neurons iteriert werden muss. Stattdessen werden in den meisten Fällen nur eine Berechnung und ein Vergleich durchgeführt. Hinzu kommen jeweils zwei Berechnungen für die Anpassung der beiden Variablen A und B, sofern ein Puls am Neuron eingetroffen ist. Hierdurch wird gegenüber der ersten Implementierungsvariante bereits die geringe Aktivität der zu simulierenden Netzwerke berücksichtigt, soweit dies im Rahmen dieser Zeitscheibensimulation möglich ist.

4.3.1.4. Messungen

Jede Netzwerkgröße wurde mit den fünf verschiedenen Aktivierungen, 4, 2, 1, 0.5 und 0.25 Spikes pro Neuron und Sekunde simuliert. Dabei feuern alle Neuronen im Netzwerk bedingt durch die randomisierten Verbindungen leicht zeitversetzt, in dem durch den Generator vorgegebenen Intervall. Aus den Messungen ist einerseits die Abhängigkeit von der Neuronenzahl, respektive der Anzahl der Verbindungen, andererseits die von der Aktivität im Netzwerk zu entnehmen.

Sowohl die GENESIS-Simulationen als auch die SPIKELAB-Simulationen wurden auf demselben Rechnersystem durchgeführt, wobei GENESIS unter Linux und SPIKELAB unter Windows NT 4.0 ausgeführt wurde. Es handelte sich dabei um ein Intel Pentium III System mit einem Prozessortakt von 500 MHz, einem Frontbustakt von 100 MHz und einem Systemspeicher von 392 MB SDRAM.

4.3.1.4.1. Varianten der Modellimplementierung in GENESIS

Die zweite, ereignisorientierte Implementierungsvariante der GENESIS-Simulation zeigt gegenüber der Standardimplementierung eine um ca. 25% geringere Simulationsdauer. In der Abbildung 43 sind die Simulationsdauern der beiden Verfahren für unterschiedlich große Netzwerke aufgetragen. Die Aktivität im Netzwerk betrug 0.25 Pulse pro Sekunde und Neuron, bei einer Simulationsgenauigkeit von 10.000 Zeitscheiben pro Sekunde. Dies zeigt deutlich, dass selbst im Rahmen der Zeitscheibensimulation Maßnahmen, welche von der geringen Aktivität der Netze profitieren, bereits zu einer merklichen Beschleunigung führen.

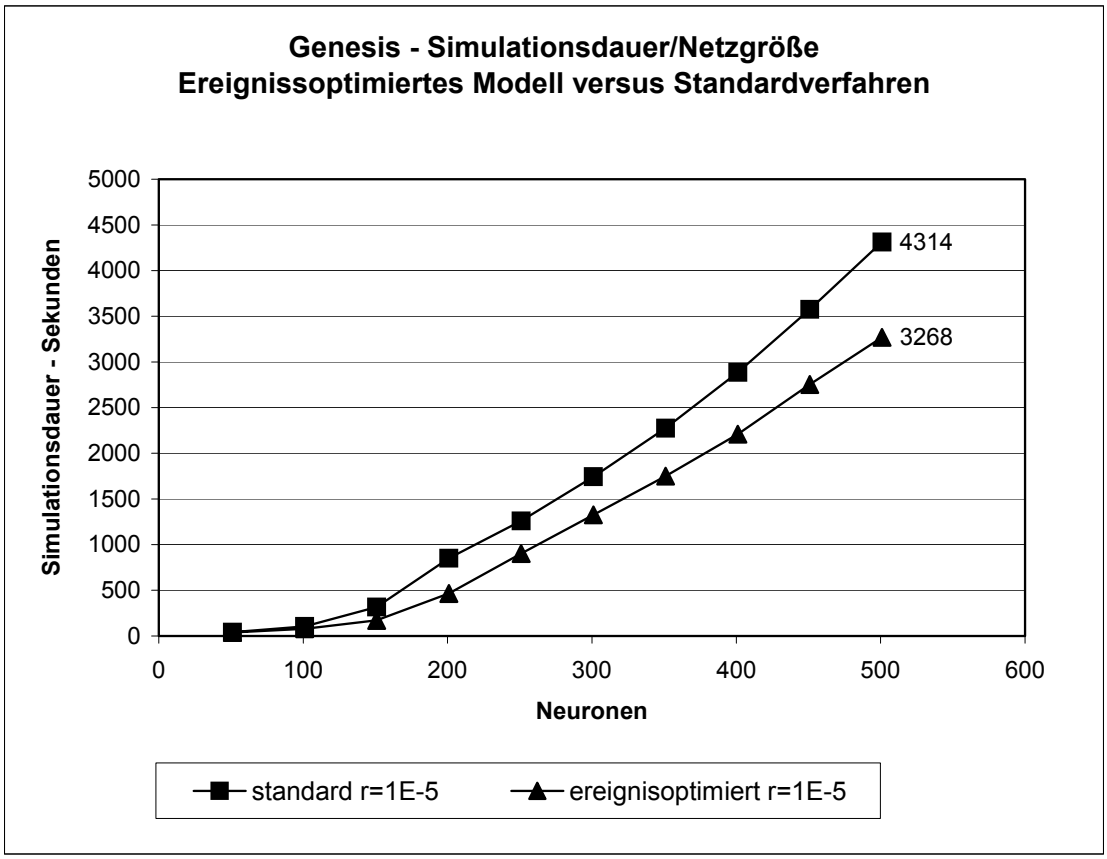


Abbildung 43: Vergleich der ereignisoptimierten GENESIS-Simulation und der Standardimplementierung (Aktivität: 0.25 Pulse pro Sekunde und Neuron, Auflösung r = 1E-5)

4.3.1.4.2. Simulationsleistung in Abhängigkeit der Aktivität im Netzwerk

Verfahrensbedingt ist die ereignisgetriebene Simulation stark abhängig von der Netzwerkaktivität, da mit steigender Aktivität auch mehr Ereignisse zu verarbeiten sind. Auf eine Zeitscheibensimulation hat die Aktivität im Netzwerk hingegen nahezu keinen Einfluss. Wie Abbildung 44 (links) zeigt, werden diese Annahmen durch die Messungen bestätigt. Zwar liegen die Simulationszeiten für die Zeitscheibensimulation bei der gewählten Auflösung von 10.000 Zeitscheiben pro Sekunde deutlich höher als die der ereignisgetriebenen Simulation, jedoch ist sie nahezu unabhängig von der Aktivität im Netzwerk. Die Kurven für die Ergebnisse der GENESIS-Simulation, bei einer eingeprägten Aktivität von 0,25 und 4 Pulsen pro Sekunde, liegen praktisch übereinander. Die Kurven für die SPIKELAB-Simulationen heben sich hingegen deutlich voneinander ab, worin sich die starke Abhängigkeit von der Aktivität im Netzwerk widerspiegelt. Für die SPIKELAB-Simulation ist zudem noch eine Messung mit einer eingeprägten Aktivität von 16 Pulsen pro Sekunde dargestellt, welche verdeutlicht, dass selbst bei einer deutlichen Anhebung der Aktivität der Abstand zu der Zeitscheibensimulation noch groß ist.

Da in der Abbildung 44 (links) zwar die eingeprägte Aktivität konstant gehalten wird, d.h. die Anzahl der Pulse, die jedes Neuron aussendet, die Zahl der Verknüpfungen jedoch überproportional im Verhältnis zu der Zahl der Neuronen ansteigt, ist die Aktivität, gemessen an der Zahl der eingehenden Pulse für größere Netzwerke, höher als für kleinere. Werden die gemessenen Simulationszeiten relativ zu diesem Anstieg umgerechnet, so ergibt sich der deutlich flachere Verlauf in Abbildung 44 (rechts).

Der verbleibende Anstieg ist darauf zurückzuführen, dass in den Simulationen mit steigender Netzgröße ein entsprechend steigender Verwaltungsaufwand entsteht. Dies gilt auch für die ereignisgetriebene Simulation.

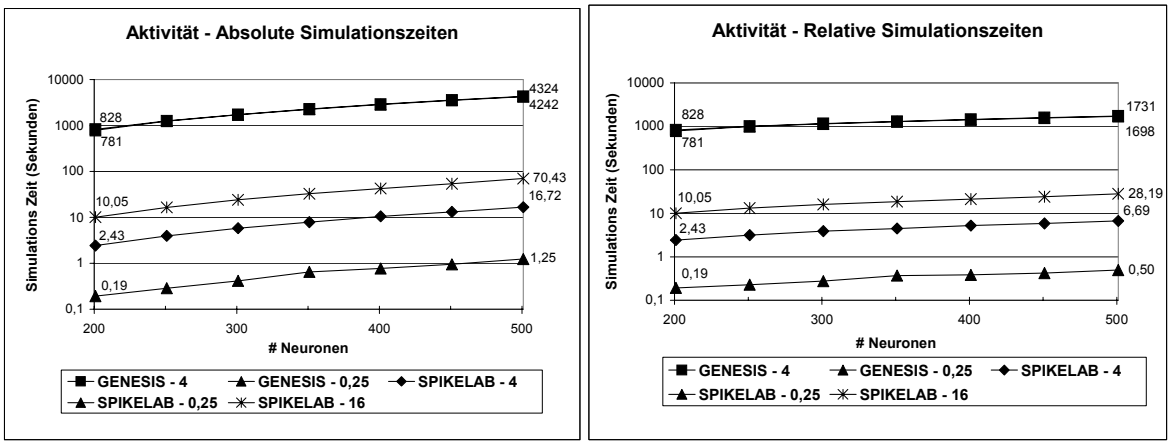


Abbildung 44: Abhängigkeit der Simulationszeiten von der Aktivität. Simulationen eines 5-lagigen Netzwerks mit 100 Neuronen pro Schicht und einer durch das Generatorneuron eingeprägten Aktivität von 0,25 und 4 Pulsen pro Sekunde in GENESIS und in SPIKELAB zusätzlich mit 16 Pulsen pro Sekunde. Links absolute Messwerte und rechts um den durch die Netzwerkgröße bedingten Anstieg der Aktivität bereinigte Messwerte.

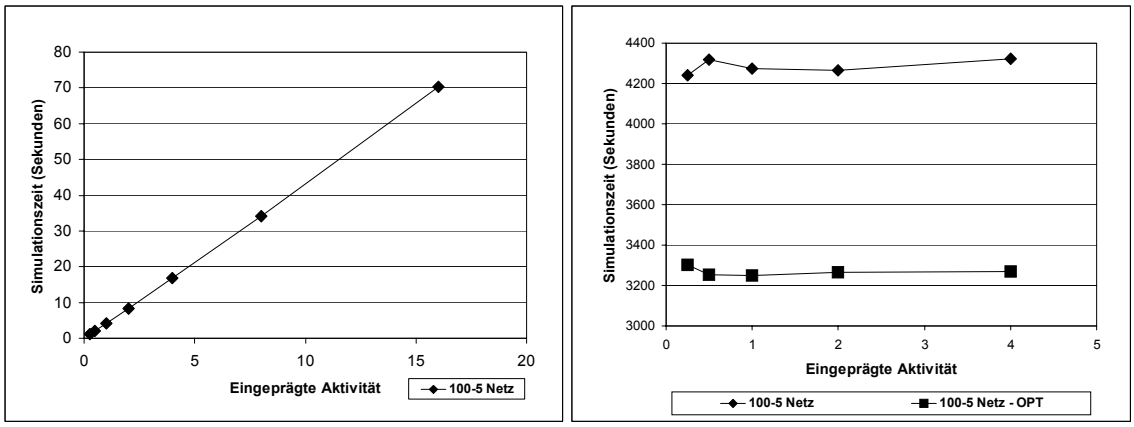


Abbildung 45: Simulationszeiten für ein 5-lagiges Netzwerk, mit 100 Neuronen pro Schicht und unterschiedlichen eingeprägten Aktivitäten – links zwischen 0,25 und 4 bzw. 16 Pulsen pro Sekunde für die SPIKELAB-Simulation und rechts zwischen 0,25 und 4 Pulsen pro Sekunde für die GENESIS Simulation (Standardverfahren und ereignisoptimiertes Verfahren) .

Da aufgrund der logarithmischen Skalierung in Abbildung 44 die Proportionalität zwischen der Netzwerkaktivität und Simulationszeit in der SPIKELAB-Simulation nicht ganz deutlich wird, ist in Abbildung 45 dieser Zusammenhang für ein 5-lagiges Netzwerk, welches 100 Neuronen pro Schicht besitzt und mit eingeprägten Aktivitäten zwischen 0,25 und 16 Pulsen pro Sekunde simuliert wurde, nochmals verdeutlicht. Es ist zu erkennen, dass eine Verdoppelung der Aktivität erwartungsgemäß auch immer zu einer Verdoppelung der Simulationszeit führt. Daneben sind für das gleiche Netzwerk die Simulationszeiten für die GENESIS-Simulation aufgetragen.

4.3.1.4.3. Genauigkeit der Simulation

Der Einfluss der Simulationsgenauigkeit wurde erfasst, indem alle Messungen mit verschiedenen Genauigkeiten, 10 µs, 100 µs und 1 ms simuliert wurden. Die Simulation in Spikelab wurde zusätzlich mit einer Simulationsgenauigkeit von 1 µs durchgeführt.

Da bei geringeren Genauigkeiten die Pulszeitpunkte unter Umständen auf die nächsten Werte auf- oder abgerundet werden, kann die Aktivität im Netzwerk absinken, obwohl der Generatorprozess immer die gleiche Pulsfolge generiert. Dieser Effekt macht sich bei der Tabellen basierten Implementierung und insbesondere bei der geringsten getesteten Auflösung von 1 ms bemerkbar, da die Ausgangspulse über die Verbindungen zwischen 1 und 3 ms zufällig verzögert werden. Abbildung 46 zeigt diesen Zusammenhang exemplarisch an einem Neuron mit drei Eingängen, wobei an jedem Eingang jeweils mit einer Millisekunde Verzögerung, ein Puls eintrifft. Die daraus entstehenden postsynaptischen Potentiale überlagern sich zu dem schwarz dargestellten Summenpotential. Wird nun der letzte Puls durch eine zu grobe Auflösung auf die nächste Millisekunde verschoben, so würde daraus die rot dargestellte Überlagerung resultieren. Im ersten Fall ergäbe sich ein Ausgangspuls, im zweiten jedoch nicht.

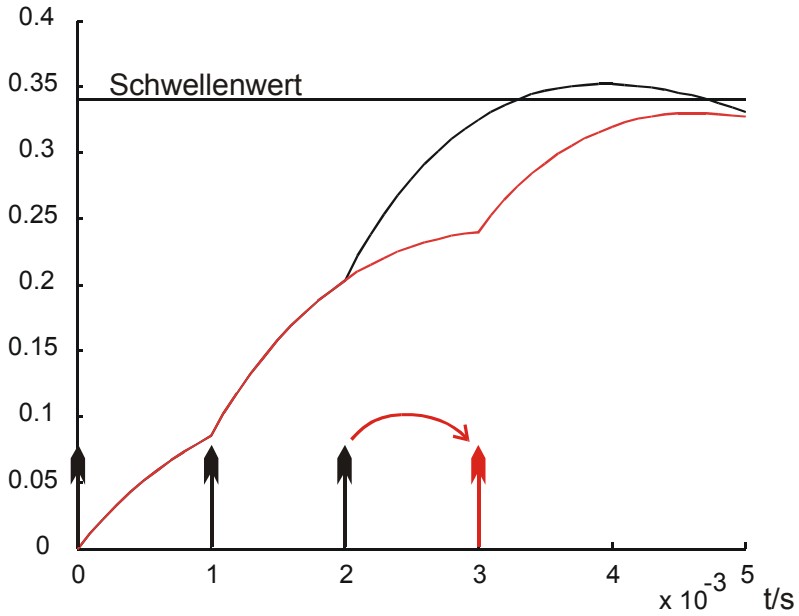


Abbildung 46: Veränderung des Summenpotentials durch ungenaue Auflösung der Pulszeitpunkte

Außerdem kann es geschehen, dass bei der Simulation mit Zeitscheiben der Schwellenwert zwischen zwei Zeitscheiben überschritten und dadurch weder von der ersten noch der zweiten Zeitscheibe erfasst wird. Denn eine Berechnung findet in der Regel immer nur zum Zeitpunkt der Zeitscheibe und nicht zwischen zwei Zeitscheiben statt. Bei der ereignisgetriebenen Simulation muss jedoch vom aktuellen Zeitpunkt aus eine Prognose für die Zukunft gestellt werden, was dazu führt, dass das Überschreiten des Schwellenwertes in der Regel erfasst und dann nur ähnlich ungenau wie in der Zeitscheibensimulation auf den nächstgelegenen auflösbaren Zeitwert gerundet wird. Diese Annahme tritt ein, wenn genauer als im Zeitscheibenabstand prognostiziert und anschließend auf die gewählte Zeitauflösung gerundet wird.

Da ein ausbleibender Puls in den folgenden Schichten ebenso zu Pulsausfällen führt, reduziert sich die gesamte Aktivität des Netzwerks aufgrund der ungenauen Simulation

deutlich. Letztlich zeigen diese Versuche, dass eine Auflösung von 1 ms zur Simulation von Neuronenmodellen mit den angegeben Zeitkonstanten unzureichend ist, da sich das gesamte Netzwerkverhalten ändert. Dies gilt zumindest für das hier untersuchte Modell, welches auf einer Tabelle basiert.

In Abbildung 47 sind die effektiven Aktivitäten für verschiedene Zeitschrittweiten der Tabellen basierten GENESIS-Simulation aufgetragen. Nur bei der höchsten getesteten Auflösung von 1E-5 Sekunden Schrittweite wird für alle Netzwerkgrößen immer die durch den Generator vorgegebene Aktivität von 4 Pulsen pro Sekunde erreicht. Bei einer Auflösung von 1 ms nimmt die effektive Aktivität deutlich ab, es gehen also einige Pulse aufgrund der Ungenauigkeit verloren. Der Effekt verstärkt sich bei kleineren Netzen, da eine geringere Streuung der synaptischen Gewichte vorliegt. Bei der ereignisgetriebenen Simulation traten bei einer Vergleichsmessung mit den Verzögerungswerten aus der Zeitscheibensimulation keine Pulsverluste auf, das heißt die Aktivität blieb auch bei der geringsten Auflösung konstant auf dem durch den *SpikeTrain* voreingestellten Wert.

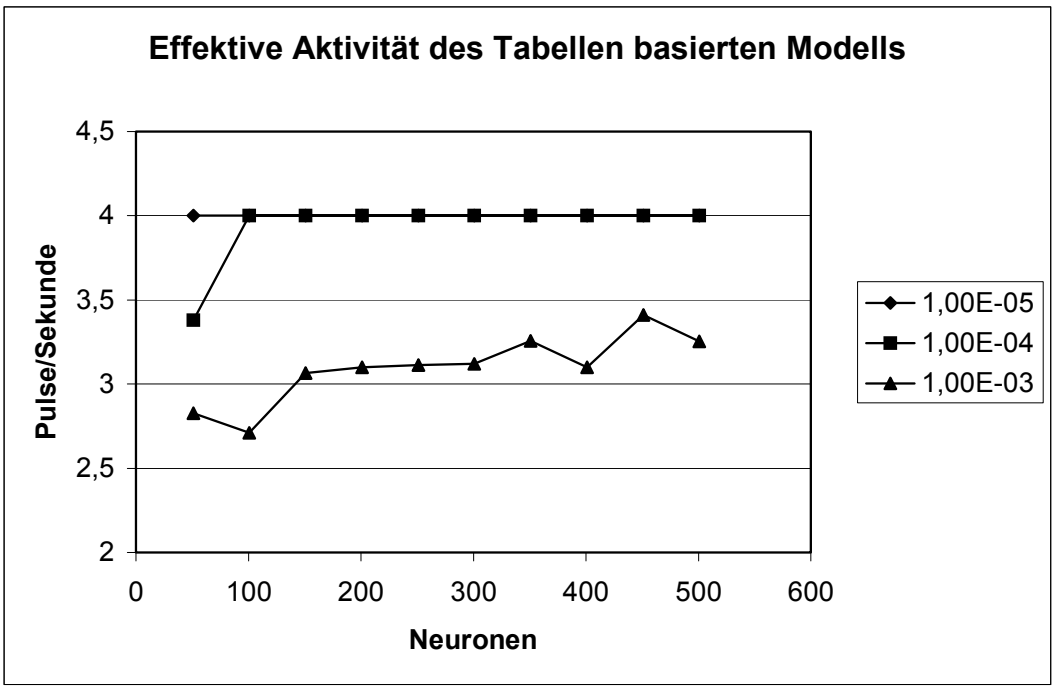


Abbildung 47: Effektive Aktivität in Abhängigkeit der zeitlichen Auflösung, bei einer eingeprägten Aktivität von 4 Pulsen pro Sekunde während einer GENESIS Simulation

Da in dem vorgestellten Neuronenmodell Zeitkonstanten gewählt wurden, welche denen aus neurophysiologischen Messungen kortikaler Neuronen entsprechen, kann aufgrund der hier angestellten Betrachtungen und Beobachtungen angenommen werden, dass die sehr häufig in Simulationen verwendete Zeitauflösung von 1 ms zur Simulation derartiger Netzwerke eher bedenklich scheint. Es liegt nicht nur eine ungenaue Bestimmung der Pulszeitpunkte vor, sondern die Aktivität im gesamten Netzwerk kann massiv beeinflusst werden. Letztlich wird also von den Tatsachen abweichendes Systemverhalten beobachtet.

Wie jedoch in Abschnitt 3.2.1 erläutert, ist die Reduktion der Auflösung letztlich das einzige Mittel, den Rechenzeitbedarf einer Zeitscheibensimulation wirklich zu reduzieren. Dieser Zusammenhang wird durch eigene Messungen mit dem vorgestellten Modell untermauert. Abbildung 48 ist zu entnehmen, dass die Simulationszeit linear mit

der Auflösung skaliert. Wird die Auflösung um eine Zehnerpotenz erhöht, so verzehnfacht sich die Simulationszeit entsprechend. Dies führt bei einer Auflösung von einer Mikrosekunde zu einer sehr hohen Simulationsdauer von schätzungsweise 12,5 Stunden pro Simulationslauf, daher wurde auf die Durchführung desselben verzichtet.

Die Simulationszeit der ereignisgetriebenen Simulation bleibt trotz Veränderungen der Auflösung nahezu gleich. Die Unterschiede sind vor allem auf Ungenauigkeiten der Messung und eine durch die Auflösung bedingte unterschiedliche Zahl von Verwaltungsereignissen während der Simulation zurückzuführen. Hierdurch wird deutlich, dass mit einer ereignisgetriebenen Simulation wesentlich genauere Simulationen möglich sind, ohne dass die Simulationszeit ansteigt.

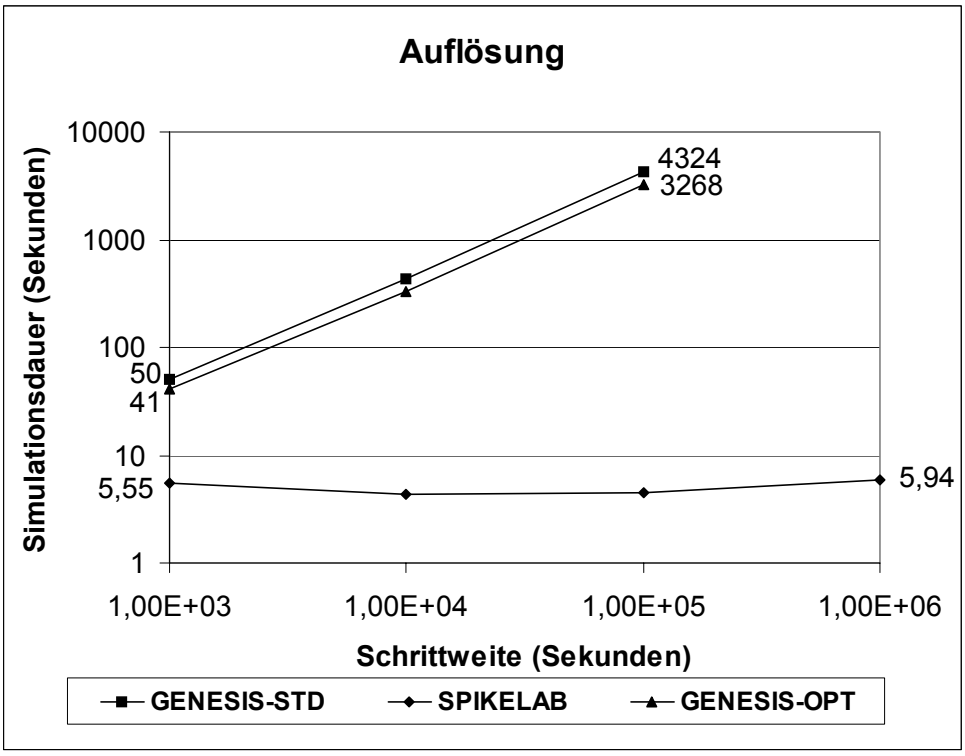


Abbildung 48: Abhängigkeit der Simulationsdauer von der zeitlichen Auflösung für ein 5-schichtiges Netzwerk mit 100 Neuronen pro Schicht (insgesamt 501 Neuronen) und einer eingeprägten Aktivität von 4 Pulsen pro Sekunde (doppelt logarithmisch skaliert)

4.3.2. Leistungscharakteristik der ereignisgetriebenen Simulation

Im Vergleich mit der Zeitscheibensimulation wurden im vorangegangenen Abschnitt die prinzipielle Abhängigkeit von der Aktivität im zu simulierenden Netzwerk sowie die Unabhängigkeit von der gewählten Zeitauflösung dargestellt. Die Leistung der ereignisgetriebenen Simulation lässt sich für unterschiedliche Netzgrößen schlecht alleine durch die Simulationszeit bestimmen, da – selbst bei konstant eingeprägter Aktivität – durch die Vernetzung eine erhöhte Aktivität in größeren Netzwerken vorliegt. Es bietet sich daher an, die Simulationsleistung in der Zahl der verarbeiteten Ereignisse oder der Zahl der verarbeiteten Pulse zu messen. Letzteres Maß gibt gewissermaßen die Nettoleistung der Simulation wieder. Mit dieser Art der Leistungsmessung lassen sich vor allem die verschiedenen Einflussfaktoren auf die Simulationsleistung auch im Vergleich mit anderen Systemen darstellen.

4.3.2.1. Leistungsmessungen

Gemessen wurden zwei verschiedene Netzanordnungen, ein geschichtetes Netzwerk (Abbildung 42) mit 5 bzw. 11 Schichten und ein vollvernetztes Netzwerk (Abbildung 49 a)). Das geschichtete Netzwerk wurde mit unterschiedlichen Schichtbreiten simuliert. Das 5-schichtige Netzwerk wurde in 10‘er Schritten in den Größen 10 bis 300 gemessen und das 11-schichtige Netzwerk wurde in 10‘er Schritten in den Größen 10 bis 200 gemessen. Bei dem voll vernetzten Netzwerk wurde die Zahl der Neuronen in 10‘er Schritten von 10 bis 500 variiert. Die größten Netzwerke markieren jeweils die maximale Netzwerkgröße, die sich mit SPIKELAB auf einem Rechner mit 256MB Speicher noch initialisieren und simulieren lässt. Alle Netzwerke wurden mit Aktivitäten zwischen 0.0625 und 16 Pulsen pro Sekunde gemessen, wobei die Aktivität vom kleinsten Wert bis zum größten von Messung zu Messung verdoppelt wurde. Bei den geschichteten Netzen wurde die Aktivität durch den speisenden Generatorprozess gesteuert und bei dem voll vernetzten Netzwerk wurden die Verzögerungen zwischen den Neuronen entsprechend eingestellt.

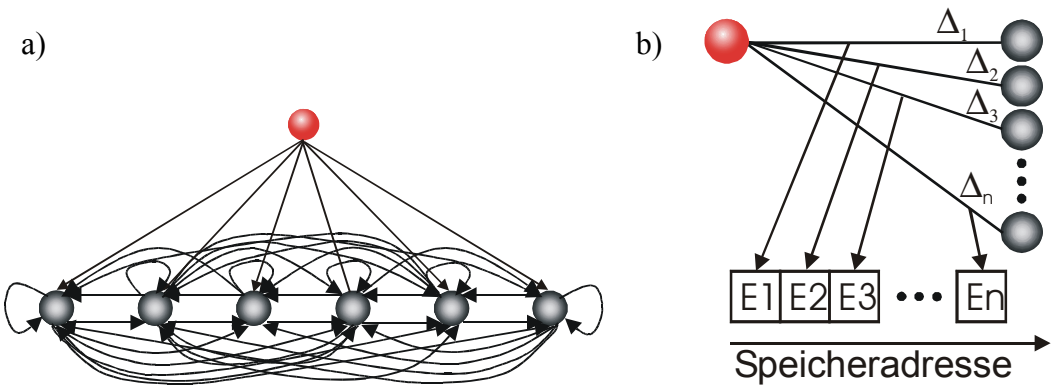


Abbildung 49: a) voll vernetztes Netzwerk b) Speicherbelegung durch Ereignisse mit $\Delta_1<\Delta_2<\Delta_3<\Delta_n$

Die beiden Netzwerktypen führen zu unterschiedlichen Auslastungen des Simulationssystems. Dies hängt vor allem von der Menge der Ereignisse ab, die während der Simulation generierten werden, und davon, wie viele sich gleichzeitig in der Ereignisliste befinden. Feuert ein Neuron, so erzeugt es dadurch so viele Ereignisse, wie es nachfolgende Neuronen in dem Netzwerk hat. Im geschichteten Netz ist dies jeweils die Schichtbreite und im vollvernetzten Netzwerk sind es alle Neuronen des Netzwerks. Durch die Art der Speicherverwaltung entstehen dabei jeweils entsprechend große Blöcke mit Ereignissen, die in diesem Block in zeitlicher Sortierung, das heißt sortiert nach den Verzögerungen abgelegt sind (Abbildung 49 b)). In der Ereignisliste werden letztlich die Speicheradressen (*pointer*) der Ereignisse eingefügt. Diese Anordnung ist für die Sortierung in der Ereignisliste optimal, da die zu verwaltenden Speicheradressen und auch die Ereignisse selber in zeitlich aufsteigender Folge im Speicher abgelegt sind. Sind nun jedoch mehrere Neuronen kurz nacheinander aktiv, so entstehen mehrere solcher Blöcke, die in der Regel auch Ereignisse mit ähnlichen Zeitstempeln aufweisen, so dass bei der zeitlichen Sortierung die Speicheradressen aus unterschiedlichen Blöcken hintereinander sortiert werden. Dies bedingt, dass relativ früh so genannte *cache misses* auftreten, d.h. es werden Daten vom Prozessor angefordert, die sich nicht im Cache befinden.

Alternativ könnten nicht die Speicheradressen, sondern die Ereignisse selber in der Ereignisliste verwaltet werden. Dies bedingt jedoch, dass die Ereignisse in die Speicherstruktur der Ereignisliste kopiert werden müssten. Da das Kopieren der Ereignisse deutlich mehr Kosten verursacht als die für den *cache* ungünstige Verwaltung der Speicheradressen in der Ereignisliste, wurde letztlich mit den Speicheradressen gearbeitet. Im vollvernetzten Netzwerk macht sich dieser Effekt besonders deutlich bemerkbar, da ein Neuron immer alle Neuronen des Netzwerks als Nachfolger hat und somit jeweils entsprechend große Blöcke entstehen würden. Bei den geschichteten Netzwerken ist die Blockgröße jedoch durch die Schichtbreite beschränkt. Abbildung 50 zeigt, wie sich dieser Zusammenhang auf die Simulationsleistung auswirkt.

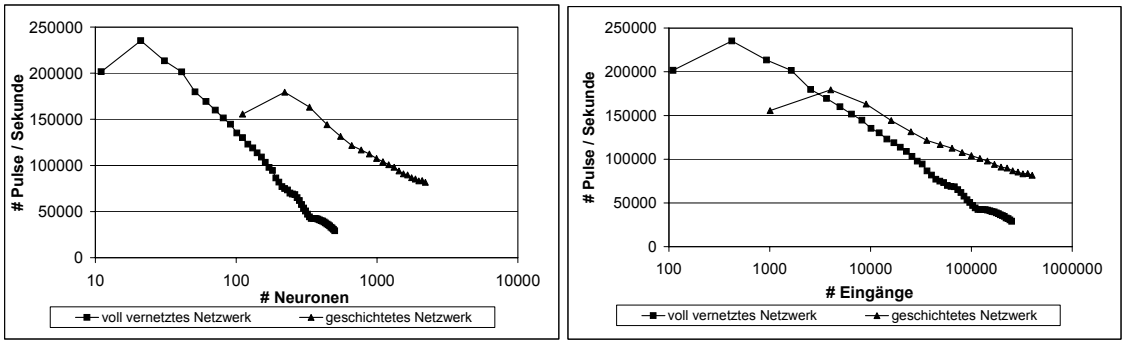


Abbildung 50: Verarbeitete Pulse pro Sekunde für ein voll vernetztes Netzwerk und ein geschichtetes Netzwerk. Links über die Zahl der Neuronen im Netzwerk und rechts über die Zahl der Eingänge im Netzwerk aufgetragen[41].

Die Simulationsleistung (Pulse pro Sekunde) sinkt bei dem voll vernetzten Netzwerk mit wachsender Größe deutlich schneller ab als bei dem geschichteten Netzwerk. Deutlicher wird dieser Zusammenhang, wenn die Kurven über die Zahl aller im Netzwerk existierenden Eingänge aufgetragen werden (rechte Seite der Abbildung). Aus dieser Gegenüberstellung wird auch deutlich, dass aufgrund des höheren Vernetzungsgrades, bei der gleichen Anzahl Neuronen, die Zahl der Eingänge im voll vernetzten Netzwerk erheblich größer ist.

In Abbildung 51 ist die maximale Blockgröße über der Netzwerkgröße für das voll vernetzte Netzwerk und das geschichtete Netzwerk (11 Schichten) aufgetragen. Der rasante Anstieg der Blockgröße beim voll vernetzten Netzwerk bedingt den in der Abbildung 50 erkennbaren stärkeren Abfall der Simulationsleistung für das voll vernetzte Netzwerk. Die maximale Blockgröße bestimmt einerseits die Datenmenge, die während eines Verarbeitungszyklus in die Ereignisliste eingefügt wird, und andererseits auch die Gesamtdatenmenge, die sich zeitweise in der Ereignisliste befindet. Wie stark dieser Abfall ausfällt, hängt einerseits davon ab, welche Ereignisliste zum Einsatz kommt, und andererseits davon, wie groß der *cache* des Systemprozessors ist, mit dem die Simulation durchgeführt wird.

[41] Es handelt sich hierbei um Simulationen auf einen Rechner mit einem 1GHz Intel Celeron Prozessor, 768 MB Speicherausbau und mit einer Aktivität von 16 Pulsen pro Sekunde.

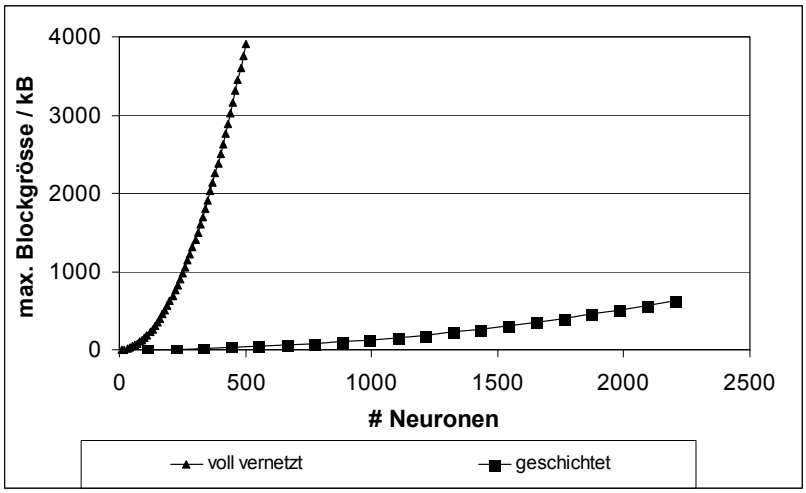


Abbildung 51: Maximale Blockgröße in Abhängigkeit der Netzgröße.

Um den Einfluss von Prozessortakt und –art zu erfassen, wurden die Netzwerke auf den drei in Tabelle 5 aufgeführten Rechnern simuliert.

System	CPU	L2 Cache	CPU-Takt	Frontbus-Takt	Speicherausbau
Rechner A	Intel Celeron	128kB	1000 MHz	110 MHz	768 MB
Rechner B	Intel Pentium 3	256kB	650 MHz	100 MHz	256 MB
Rechner C	Intel Pentium 3	256kB	500 MHz	100 MHz	392 MB

Tabelle 5: Eckdaten der zur Simulation verwendeten Rechnersysteme

Die aufgeführten Systeme unterscheiden sich einerseits aufgrund der Prozessortaktrate und andererseits weist Rechner A gegenüber B und C einen kleineren *cache* auf.

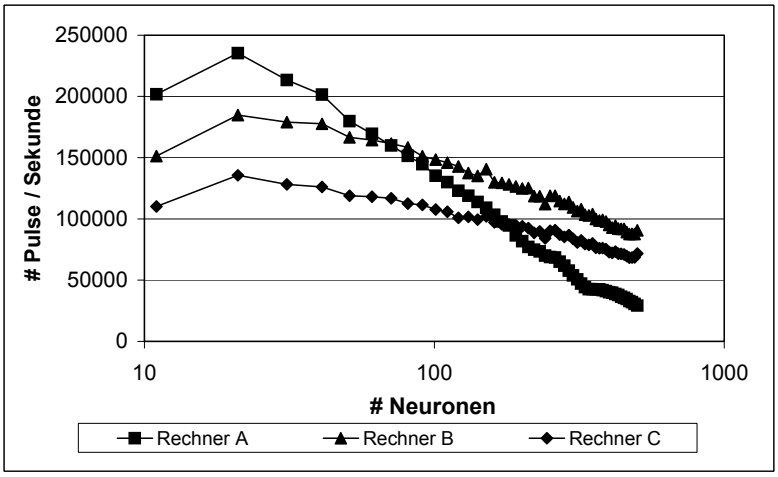


Abbildung 52: Einfluss der *cache*-Größe auf die Simulationsleistung (voll vernetztes Netzwerk)

Wird die gleiche Ereignisliste verwendet, so zeigt sich in Abbildung 52 für die Rechner B und C ein flacherer Abfall der Simulationsleistung, der letztlich auf den größeren *cache* zurückzuführen ist. Untermauert wird diese Annahme durch die gleichartigen Verläufe von Rechner B und C, die nur aufgrund ihrer unterschiedlichen Taktrate insgesamt zueinander verschoben sind.

Bei dem geschichteten Netzwerk macht sich der Cache bei weitem nicht so stark bemerkbar, da hier die maximale Blockgröße nicht so anwächst wie beim vollvernetzten Netzwerk. Auch die Gesamtdatenmenge, die von der Ereignisliste gehalten werden muss, ist nicht so hoch wie bei dem voll vernetzten Netzwerk. In Abbildung 53 ist zu erkennen, dass sich der *cache* zwar bemerkbar macht, da Rechner B mit einer um 35% geringeren Prozessortaktrate als Rechner A für größere Netzwerke eine vergleichbare oder zum Teil höhere Simulationsleistung erreicht, jedoch ist dieser Effekt nicht so deutlich wie bei dem vollvernetzten Netzwerk. Auch für diesen Netzwerktyp zeigt die Messkurve von Rechner C den gleichen, lediglich insgesamt verschobenen Verlauf wie die Messkurve von Rechner B.

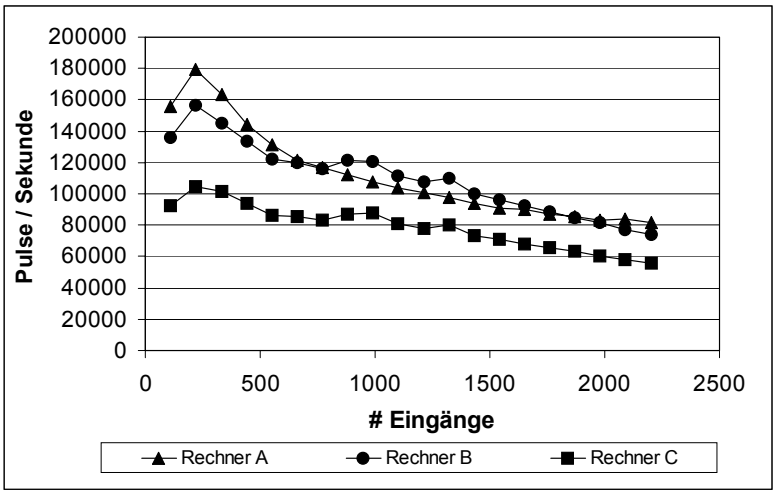


Abbildung 53: Einfluss der *cache*-Größe auf die Simulationsleistung (geschichtetes Netzwerk)

4.3.2.2. Ereignislisten

In Abschnitt 3.1.1.2.4 wurden Alternativen zur Implementierung der Ereignisliste diskutiert. Es stehen in SPIKELAB wahlweise zwei Ereignislistenalgorithmen zur Verfügung, eine auf einer STL[42] *priority queue* und eine *calendar queue* basierte Implementierung.

Die STL *priority queue* Implementierung basiert auf einem binären Baum, die *calendar queue* ist in Abschnitt 3.1.1.2.4 näher ausgeführt. Aus der Abbildung 54 wird deutlich, dass für kleine Netzwerkgrößen die *priority queue* kürzere Simulationszeiten erreicht, wohingegen die *calendar queue* bei größeren Netzen und damit potentiell höheren Füllständen der Ereignisliste die kürzeren Simulationszeiten aufweist. Ist eine größere Gesamtdatenmenge zu verwalten, so bietet sich die *calendar queue* an (vgl. Abschnitt 3.1.1.2.4 – Abbildung 19), da die Einfüge- und Entnahmeoperationen deutlich schneller vonstatten gehen als bei der *priority queue* aus der STL. Die schwache Leistung bei kleineren Netzwerken ist einerseits auf die Speicherverwaltung der vorliegenden Ereignislistenimplementierung [Werkhausen2002] zurückzuführen, andererseits auf einen größeren Overhead vor allem bei der Entnahme von Ereignissen aus der Ereignisliste. Der Leistungseinbruch für die größten Netzwerke ist darauf zurückzuführen, dass die *calendar queue* Implementierung auf dem Container *vector* aus der STL basiert (Abbildung 54: cq-vec-16). Dieser Container verdoppelt seine Größe, sobald die zuvor

[42] **S**tandard **T**emplate **L**ibrary

gesetzte Kapazitätsgrenze erreicht ist, und belegt damit in der Regel mehr Speicher als für die Ablage der Ereignisse notwendig wäre. Dies provoziert somit das Auslagern von Daten, wenn größere Datenmengen zu verwalten sind. Zum Vergleich wurde eine Messung mit dem Container *list* aus der STL als zugrunde liegende Datenstruktur durchgeführt; da in diesem Container die Daten dynamisch verwaltet werden, tritt hier erwartungsgemäß kein Einbruch bei den großen Netzwerken auf. Jedoch reduziert sich die Gesamtleistung insbesondere für die mittleren Netzgrößen. Gelingt es, diese Defizite der *calendar queue* Implementierung in zukünftigen Versionen zu beheben, so bietet sich die Verwendung der *calendar queue* durchweg gegenüber der *priority queue* an, da sie schon für mittlere Netzwerkgrößen durchschnittlich eine 10% höhere Simulationsleistung erzielt.

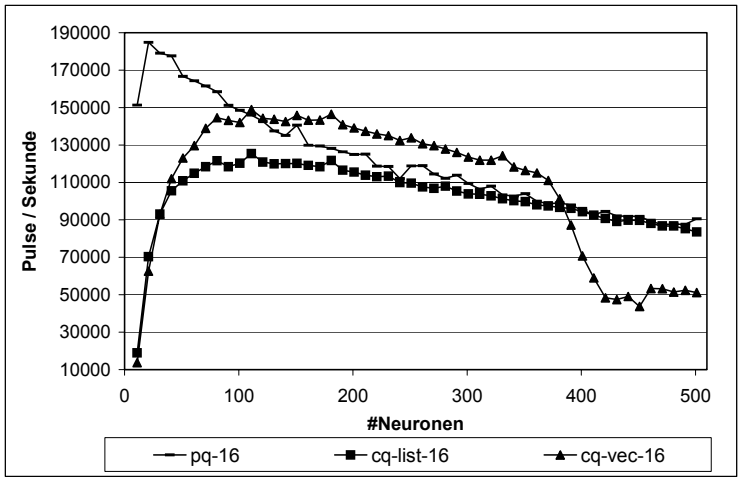


Abbildung 54: Simulationsleistung mit *priority queue* (pq) und *calendar queue* (cq), basierend auf dem STL *vector* (vec) und der STL *list* (list) - vollvernetztes Netzwerk auf Rechner B, Aktivität 16 Pulse pro Sekunde

Beide Ereignislisten weisen verschiedene Charakteristiken auf, welche insbesondere vom Füllstand der Liste und der Menge der Elemente abhängen, die gleichzeitig eingefügt werden. Daher wirken sich Optimierungen, bei denen mehrere Ereignisse zur Verarbeitung zusammengefasst werden, auch recht unterschiedlich aus. Zudem stellt die Aktivität im Netzwerk eine ähnliche Einflussgröße dar.

4.3.2.3. Gleichzeitige Verarbeitung mehrerer Ereignisse

In Abschnitt 3.3.1 wurde dargestellt, dass bei der ereignisgetriebenen Simulation eine Beschleunigung erreicht werden kann, wenn mehrere Ereignisse der Ereignisliste entnommen werden, um sie gleichzeitig zu verarbeiten. Die Beschleunigung darf auf eine reduzierte Zahl von Kontrollereignissen zurückgeführt werden (vgl. Abschnitt 3.3.1). Abbildung 55 zeigt den prozentualen Mehraufwand der ereignisgetriebenen Simulation mit und ohne gebündelte Verarbeitung der Ereignisse für eine Aktivität von 16 Pulsen pro Sekunde und Neuron. Der Mehraufwand lässt sich durch das Verhältnis zwischen der Gesamtzahl aller verarbeiteten Ereignisse relativ zur Zahl der, im betreffenden Simulationslauf zu verarbeiteten Pulse ausdrücken. Aus Abbildung 55 ist zu entnehmen, dass mit zunehmender Zahl verarbeiteter Pulse der Mehraufwand drastisch absinkt, wohingegen der Mehraufwand bei der Standardverarbeitung nahezu konstant bleibt. Auf der Abszisse ist die Netzwerkgröße aufgetragen, mit der auf Grund der konstant eingeprägten Aktivität von 16 Pulsen pro Neuron und Sekunde größere Netzwerke auch immer eine proportional größere Menge zu verarbeitender Pulse bedeutet. Diese Verläufe

entstehen, da die Zahl der „Verwaltungsereignisse“ bei der Mehrfachverarbeitung nahezu konstant bleibt, bei der Standardverarbeitung jedoch genauso proportional zur Netzwerkgröße wächst, wie die Zahl der zu verarbeitenden Pulse. Weitere Messungen zeigten, dass der Betrag und der Verlauf des Mehraufwands weitgehend unabhängig von der Aktivität im Netzwerk sind.

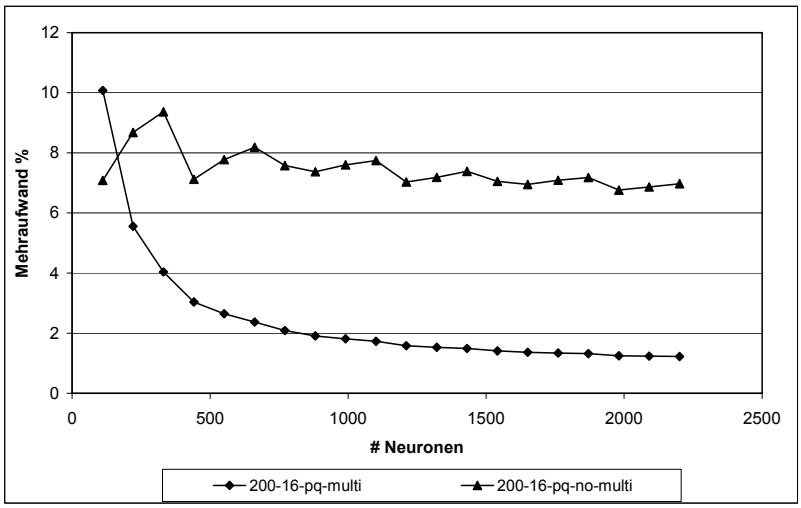


Abbildung 55: Prozentualer Mehraufwand der ereignisgetriebenen Simulation mit und ohne gebündelte Verarbeitung für ein Netzwerk mit 11 Schichten und einer Aktivität von 16 Pulsen pro Sekunde auf Rechner B.

Dennoch wirkt sich die Reduktion der „Verwaltungsereignisse“ nicht im gleichen Maße auf die Simulationsleistung aus, da insbesondere die Ereignislistenalgorithmen, abhängig von der Füllung und der Zahl einzufügender Ereignisse, also abhängig von der Aktivität im Netzwerk, unterschiedliche Laufzeiten aufweisen (vgl. Abschnitt 3.1.1.2.4). In Abbildung 56 ist für ein Netzwerk mit 11 Schichten die Simulationsleistung mit und ohne gebündelte Verarbeitung von Ereignissen über die Netzwerkgröße aufgetragen. Da die Messkurven für eine Aktivität von 0,25 Pulsen pro Sekunde praktisch identisch sind, ist für beide Fälle nur eine Kurve dargestellt.

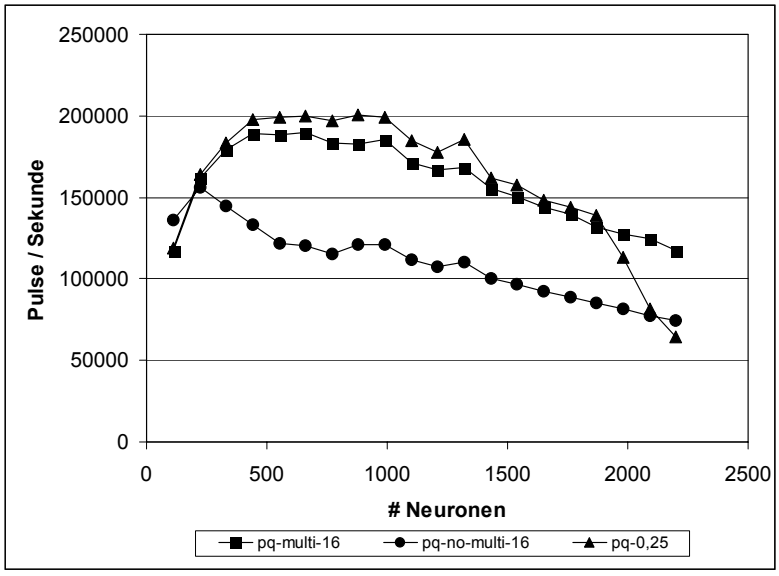


Abbildung 56: Simulationsleistung mit und ohne gebündelte Verarbeitung für ein Netzwerk mit 11 Schichten und einer Aktivität von 16 bzw. 0,25 Pulsen pro Sekunde auf Rechner B, *priority queue*.

Bei einer Aktivität von 16 Pulsen pro Sekunde zeigt sich jedoch ein deutlicher Unterschied in der Simulationsleistung, abhängig davon, ob eine gebündelte Verarbeitung stattfindet oder nicht. Mit der gebündelten Verarbeitung gelingt es bei höheren Aktivitäten nahezu die Leistung zu erreichen, die sonst nur mit deutlich geringeren Aktivitäten erreicht werden kann.

Wird die *calendar queue* als Ereignisliste verwendet, so zeigt sich dieser Effekt noch weitaus deutlicher, da hier mit wachsender Aktivität und dadurch wachsender Zahl der Ereignisse die Leistung der *calendar queue* ansteigt. In Abbildung 57 liegen nun die beiden Kurven für eine Aktivität von 0,25 Pulsen pro Sekunde weiter auseinander, da sich bereits bei dieser Aktivität für große Netzwerke ein Einfluss der gebündelten Verarbeitung bemerkbar macht. Bei einer Aktivität von 16 Pulsen pro Sekunde profitiert die Simulation mit der *calendar queue* als Ereignisliste deutlich von dem reduzierten Mehraufwand, so dass in diesem Fall die maximale Simulationsleistung mit der gebündelten Verarbeitung erreicht wird.

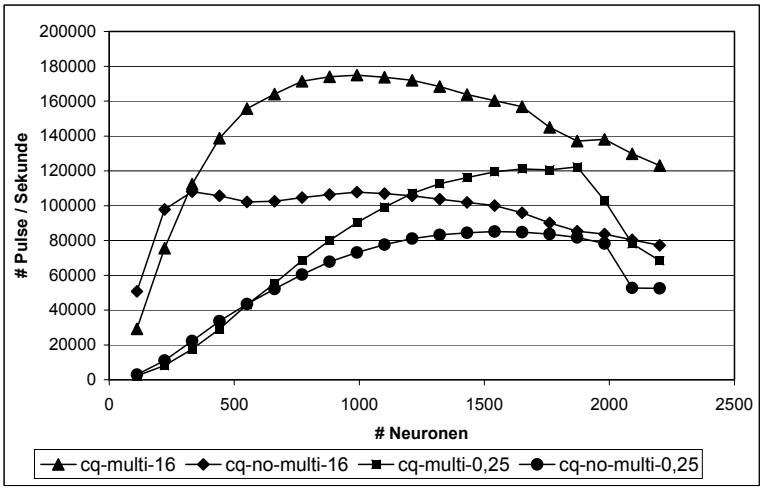


Abbildung 57: Simulationsleistung mit und ohne gebündelte Verarbeitung für ein Netzwerk mit 11 Schichten und einer Aktivität von 16 bzw. 0,25 Pulsen pro Sekunde auf Rechner B, *calendar queue*.

4.3.2.4. Ergebnisse und Folgerungen

Aus den präsentierten Messergebnissen wird deutlich, dass die ereignisgetriebene Simulation stark von dem verwendeten Ereignislistenalgorithmus abhängt. Dies liegt einerseits an der beträchtlichen Anzahl von Ereignissen, die dort verwaltet werden müssen und andererseits an der hohen Bedarf an Speicherbandbreite, die für die generierten Ereignisse bereitgestellt werden muss. Obwohl die Simulationsleistung deshalb für größere Netzwerke abnimmt, lassen sich Bereiche nahezu konstanter Simulationsleistung feststellen. Aus der Abbildung 58 links ist zu entnehmen, dass auf dem Rechner B bis zu einer Schichtbreite von 130 Neuronen pro Schicht das geschichtete Netzwerk mit einer relativ konstanten Simulationsleistung simuliert wird, erst danach fällt die Simulationsleistung stetig ab. Dies gilt für Simulationen mit und ohne gebündelte Verarbeitung, wobei die Simulationsleistung bei der gebündelten Verarbeitung auf einem höheren Niveau verläuft. Für das voll vernetzte Netzwerk lässt sich nur bei der gebündelten Verarbeitung ein nahezu konstanter Verlauf bis zu einer Zahl von 121 Neuronen feststellen (Abbildung 58 rechts), bevor ein deutlicher Abfall der Simulationsleistung stattfindet. Ohne gebündelte Verarbeitung fällt die Simulationsleistung von Anfang an ab, was aus den zuvor geschilderten Eigenschaften dieses Netzwerks herrührt.

Wird die *calendar queue* verwendet, so beginnen die Bereiche konstanter Simulationsleistung später, jedoch erstrecken sich diese dann hin zu größeren Netzwerken. In Abbildung 59 ist dieser Zusammenhang anhand der Simulationsleistung von Rechner C für das geschichtete Netzwerk links und das vollvernetzte Netzwerk rechts dargestellt.

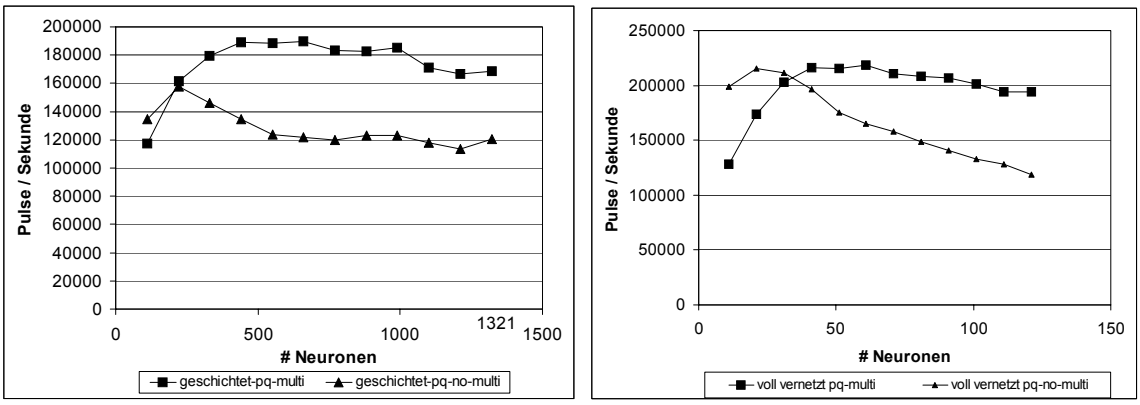


Abbildung 58: Bereiche konstanter Simulationsleistung: Links geschichtetes Netzwerk, rechts voll vernetztes Netzwerk auf Rechner B, jeweils mit und ohne gebündelte Verarbeitung, *priority queue*.

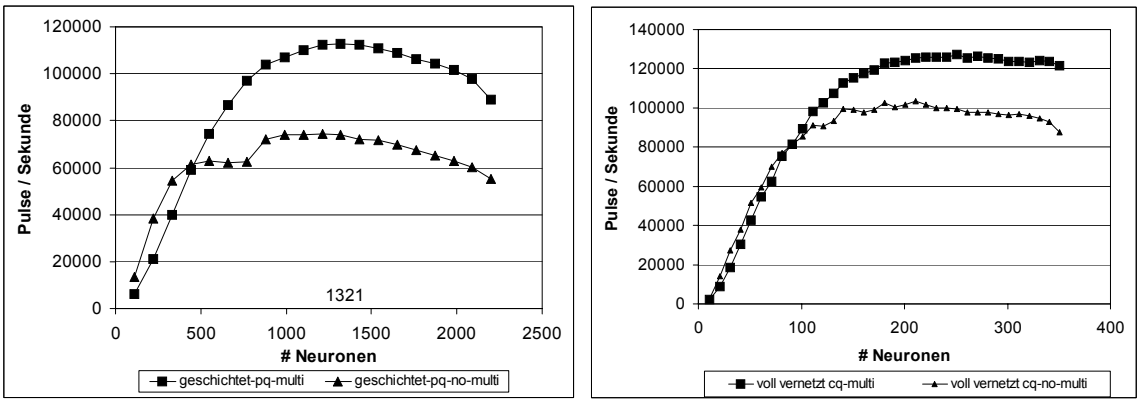


Abbildung 59: Bereiche konstanter Simulationsleistung: Links geschichtetes Netzwerk, rechts voll vernetztes Netzwerk auf Rechner C, jeweils mit und ohne gebündelte Verarbeitung, *calendar queue*.

Die Ausdehnung der Bereiche konstanter Simulationsleistung hängt letztlich von dem verwendeten Ereignislistenalgorithmus, der Größe des *caches* und dem Speichersubsystem (Frontbustakt und Speicherausbau) des verwendeten Rechners ab.

Der Vergleich mit einer nicht spezialisierten Zeitscheibensimulation in GENESIS hat deutlich die Vorteile der ereignisgetriebenen Simulation für die betrachteten Netzwerke aufgezeigt. Damit konnte die grundsätzliche Annahme, dass ein ereignisgetriebenes Verfahren für derlei Netzwerke effizienter ist, untermauert werden. Es sei an dieser Stelle noch angemerkt, dass die gewählten Testnetzwerke entweder eine maximale (vollvernetzte Netzwerke) oder eine recht dichte (schichtweise vollvernetzte Netzwerke) Vernetzung aufweisen und damit nicht den für die ereignisgetriebene Simulation günstigsten Fall einer extrem geringen Vernetzung abdecken. Die Wahl der Testnetze erfolgte jedoch auch im Hinblick auf Netzwerke, die zur Lösung von technischen Problemen eingesetzt werden. Hier findet man recht häufig zumindest schichtweise vollvernetzte Netzwerke und sehr selten noch spärlicher vernetzte Netzwerke.

Darüber hinaus stellt sich jedoch die Frage, wie das Leistungsverhältnis zwischen dem vorgestellten Simulationsverfahren und dem spezialisierten Zeitscheibenverfahren aus-

fällt. Solche Simulationsverfahren arbeiten zum Teil mit wesentlichen Einschränkungen, wie einer festen Zeitauflösung von 1 ms und dem Verzicht, Verzögerungen zu modellieren.

Um einen Vergleich zu ermöglichen sind in Tabelle 6 Messwerte aus [Wolff2001] aufgeführt, die für ein Netzwerk mit insgesamt 114.401 Neuronen und 3.764.120 synaptischen Verbindungen auf unterschiedlichen Rechnern ermittelt wurden. Das Netzwerk generiert laut [Wolff2001] in jeder Millisekunde im Durchschnitt 28.269 Eingangspulse. Damit lässt sich das in dieser Arbeit verwendete Maß der verarbeiteten Pulse pro Sekunde ableiten (Tabelle 6). Da die erzielte Simulationsleistung des Simulationsverfahrens nach [Wolff2001] für vergleichbare Plattformen maximal doppelt so hoch ist wie die des vorgestellten Simulationsverfahrens in SPIKEALB, erreicht das ereignisgetriebene Verfahren in SPIKELAB eine vergleichbare Leistung.

Maschine	**Bsp. Netz Zeit in ms**	**SPEC INT in ms**	**SPEC FP in ms**	**Bsp. Netz Pulse/Sekunde**
Sun Sparc 5 / 110	235	1,59	1,99	111.735
Sun Sparc 20 / 71	190	3,11	3,1	148.784
Sun Ultra 1 / 170	108	6,26	9,06	261.750
Sun Ultra 2 / 200	85	7,81	14,7	332.576
Sun Ultra 5 / 270	80	9,17	10,6	353.363
Sun Ultra 60 / 360	45	16,1	23,5	628.200
PC Pentium II / 266	85	10,7	8,17	332.576
DEC Alpha 250/266	140	4,18	5,78	201.921

Tabelle 6: Leistungsmessungen aus [Wolff2001] mit einem Netzwerk, welches im Mittel 28.269 Eingangspulse pro ms generiert – hier ergänzt um die Zahl der verarbeiteten Pulse pro Sekunde.

Bei diesem Vergleich offenbart sich jedoch ein entscheidender Nachteil der vorliegenden Implementierung, der nicht in jedem Fall auf das Verfahren selber zurückzuführen ist: Die Netzwerkgrößen, die simuliert werden können, sind im Vergleich zu den spezialisierten Zeitscheibenverfahren deutlich geringer. Dies beruht einerseits auf dem Verfahren selber, da jede Verbindung eine eigene Verzögerung aufweist und individuell behandelt wird, andererseits wäre es möglich, die vorliegende Implementierung auf die Simulation solcher großen Netzwerke hin zu optimieren, indem z. B. nicht explizit gespeicherte Verbindungen, sondern zur Laufzeit berechnete Verbindungen verwendet werden. Außerdem ergibt sich durch eine stärker vereinfachte Modellierung der Ereignisse im System ein weiteres Optimierungspotential im Hinblick auf einen geringeren Speicherbedarf der Simulation.

Bereits ohne derlei Optimierungen kann daher das vorgestellte Verfahren von seiner Leistungsfähigkeit nahe bei den spezialisierten Zeitscheibenverfahren angesiedelt werden, wobei lediglich die Größe der zu simulierenden Netzwerke beschränkt ist. Für große Netzwerke ist dann eine Verteilung der Simulation erforderlich, die, wie im folgenden Abschnitt zu sehen ist, zudem zu einer Beschleunigung der Simulation führt.

4.3.3. Leistung der verteilten Simulation

4.3.3.1. Übertragungsleistung

In Abschnitt 3.4.6 wurden Messungen der Latenz und der Bandbreite unterschiedlicher Übertragungsverfahren vorgestellt. Da die Übertragungsleistung innerhalb einer verteilten Simulation sich exakt so verhält wie das darunter liegende Nachrichtensystem, kann auf diese Messungen zurückgegriffen werden. Lediglich der Aufwand, der neben dem reinen Transfer in der verteilten Simulation gegenüber einer Simulation auf einem einzelnen Rechner hinzukommt, muss noch bestimmt werden. Hierfür kommt ein sehr einfaches Testnetzwerk zum Einsatz. Das Netzwerk zeigt ein ähnliches Verhalten wie die in Abschnitt 3.4.6 verwendete Echoapplikation. Ein Generatorprozess speist in einem regelmäßigen Intervall Ereignisse in einen Prozess, welcher die Ereignisse lediglich weiterleitet (Klasse *IdProcess*). Der Ausgang des Prozesses ist wiederum mit einem solchen *IdProcess* verbunden. Für eine verteilte Version wird nun der erste *IdProcess* auf den zweiten Rechner verlagert, so dass wie in der Echoapplikation ein generiertes Ereignis zu dem ausgelagerten *IdProcess* geschickt wird und von diesem wieder zurück an den zweiten *IdProcess*, also, wie in der Echoapplikation aus Abschnitt 3.4.6, zwei Sende- und Empfangsvorgänge stattfinden. Um nur die Übertragungsleistung innerhalb der Simulation zu messen, wurde der Generator so eingestellt, dass in jeder Mikrosekunde ein Ereignis generiert wurde. Dies führt dazu, dass permanent eine Übertragung stattfindet und die in Abschnitt 4.2.3 beschriebenen Verfahren zur Blockadeerkennung und –auflösung nicht aufgerufen werden.

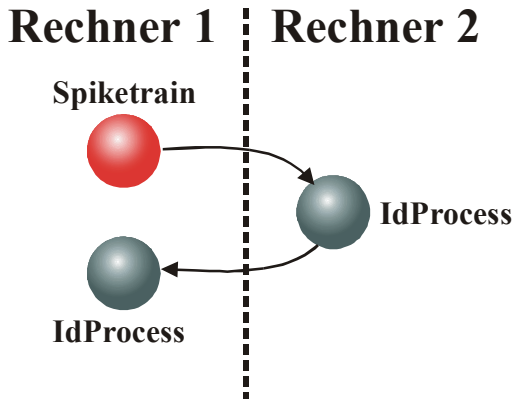


Abbildung 60: Testnetz zur Bestimmung der Übertragungsleistung in der verteilten Simulation.

Das Testnetzwerk wurde auf PCs mit Windows NT und CORBA als Nachrichtensystem simuliert. Für die Einzelplatzsimulation kam ein PC mit Pentium-III Prozessor und einer Taktrate von 650 MHz sowie 256MB RAM (100 MHz Frontbustakt) zum Einsatz. Für die verteilte Simulation wurde zusätzlich ein PC mit Pentium-III Prozessor und einer Taktrate von 500MHz und 392MB RAM (100 MHz Frontbustakt) eingesetzt. Die beiden Rechner wurden über ein 100Mbit Ethernet Switch verbunden. Da der Subsimulator, welcher lediglich einen *IdProcess* simulieren muss, eine geringere Rechenleistung als der Subsimulator mit *SpikeTrain* und *IdProcess* benötigt, wurde dieser auf dem leistungsschwächeren Rechner ausgeführt. Durch diese Maßnahme fällt der Leistungsunterschied der Rechner nicht ins Gewicht[43].

[43] Der Subsimulator verursachte auf dem zweiten Rechner durchschnittlich eine 50% Auslastung der CPU.

Die Simulation des beschriebenen Netzwerks beanspruchte für eine Sekunde simulierter Zeit auf der Einzelplatzversion des Simulators 28 Sekunden und in der verteilten Simulation 262 Sekunden[44]. Die durch die Verteilung bedingte Transferzeit betrug somit 234 Sekunden. Da in einer Sekunde simulierter Zeit 1 Millionen Ereignisse zu je 16 Byte hin- und zurückgeschickt wurden, benötigte ein Hin- und Rücktransfer 234 µs. Verglichen mit der Messung für den reinen Datentransfer, bei dem durchschnittlich 212 µs gemessen wurden, lag dieser Wert um 22 µs höher. Die Simulatorklassen verursachten demnach einen Anstieg der Transferzeit um ca. 10% bezogen auf den reinen Datentransfer. Da die Messungen unter Laborbedingungen und nicht in einem Netzwerk mit den üblichen Lastschwankungen stattfanden, lassen die Messwerte lediglich eine Aussage über den maximal erreichbaren Durchsatz zu.

Da der Anstieg der Transferzeit unabhängig von der darunter liegenden Verbindung ist, ergeben sich für langsamere Verbindungen wie SUN-SUN und PC-SUN rechnerisch entsprechend geringere prozentuale Steigerungen der Transferzeiten (Tabelle 7). Für Transfers mit mehreren Ereignissen lassen sich die Übertragungszeiten aus den Messungen in Abschnitt 4.2.3 ableiten.

Verbindung	nur Daten(µs)	Simulator (µs)	Rel. Steigerung (%)
CORBA PC_PC B.	212	234	10
CORBA SUN_SUN B.	374	396	6
CORBA PC_SUN B.	1155	1177	2

Tabelle 7: Relative Erhöhung der Transferzeit innerhalb des Simulators, verglichen zur Messung des Datentransfers ohne Simulator. (B. steht für blockierendes Senden/Empfangen)

4.3.3.2. Leistungsmessungen mit Testnetzwerken

Die Testnetzwerke, welche in Abschnitt 4.3.2 vorgestellt wurden, werden auch zur Bewertung in der verteilten Simulation herangezogen. Das voll vernetzte Netzwerk stellt dabei den ungünstigsten Fall für die Verteilung dar, da unabhängig von der gewählten Zerlegung des Netzwerks immer eine maximale Kommunikation stattfindet. Dagegen ist das geschichtete Netzwerk stärker entkoppelt, da in einer Schicht bereits mit der Berechnung neuer Ereignisse begonnen werden kann, selbst wenn die nachfolgende Schicht noch mit der Berechungen der Ergebnisse aus dem vorangegangenen Berechnungsschritt beschäftigt ist. Dies ist darauf zurückzuführen, dass es keine rückführenden Verbindungen von höheren zu niedrigeren Schichten gibt. Aufgrund der Netzwerktopologie kann also gewissermaßen eine Verarbeitungspipeline aufgebaut werden. Das geschichtete Netz stellt somit einen recht günstigen Fall für die Verteilung der Simulation dar. Es darf angenommen werden, dass sich alle anderen Netzwerktopologien bezüglich der Skalierbarkeit zwischen diesen beiden Extremen bewegen. Nur in Spezialfällen, in denen eine weitaus stärkere Entkopplung von einzelnen Bereichen eines Netzwerks vorliegt, könnten noch bessere Werte als beim geschichteten Netzwerk durch eine Verteilung erreicht werden. Dies würde der Simulation verschiedener Hirnareale entsprechen, die nur eine sehr lose Kopplung aufweisen.

Zur Simulation wurden die in Abschnitt 4.3.2.1 beschriebenen Rechnersysteme verwendet und zusätzlich die in Tabelle 8 aufgeführten Systeme.

[44] Bei umgekehrter Zuordnung der Netzwerkteile wurden folgende Zeiten gemessen: 267s für die verteilte Simulation und 37s für die Einzelplatzsimulation auf dem leistungsschwächeren Rechner.

System	CPU	L2 Cache	CPU-Takt	Frontbus-Takt	Speicherausbau
Rechner D	Intel Pentium 3	256 kB	700 MHz	100 MHz	256 MB
Rechner E	Intel Pentium 3	256 kB	750 MHz	100 MHz	256 MB
Rechner F	Intel Pentium 3	256 kB	800 MHz	133 MHz	256 MB
Rechner G	Intel Pentium 3	256 kB	1000 MHz	133 MHz	256 MB

Tabelle 8: Kenndaten der Rechner, auf welche die Simulation verteilt wurde.

Für Messungen zu diesen Untersuchungen standen dauerhaft nur die Rechnersysteme A, B und C zur Verfügung, welche in einem lokalen 100Mbit-Netzwerk mit Sterntopologie über einen Switch gekoppelt waren. Um die Skalierbarkeit der Simulation zu verdeutlichen, wurden abschließend noch Messungen mit den Rechnern B, D, E, F und G durchgeführt. Diese Rechner sind Bestandteil eines großen lokalen Netzwerks, welches eine Sterntopologie aufweist, in der die Rechner über 100Mbit Switches miteinander gekoppelt sind. Da die Messungen mit den Rechnern B, D, E, F und G in einem großen lokalen Netzwerk stattfanden, stand für diese Messungen nicht wie bei den Messungen mit den drei Rechnern A bis C die gesamte Bandbreite des Netzwerks zur Verfügung. Außerdem wiesen die Rechner B, D, E, F und G eine andere Leistungscharakteristik als die Rechner A, B und C auf. Um dennoch einen Vergleich zwischen den Messergebnissen zu ermöglichen, wurden die Messergebnisse für die verteilte Simulation einerseits auf das Minimum und andererseits auf den Mittelwert aller Simulationsdauern aus den Einzelplatzsimulationen der beteiligten Rechner bezogen.

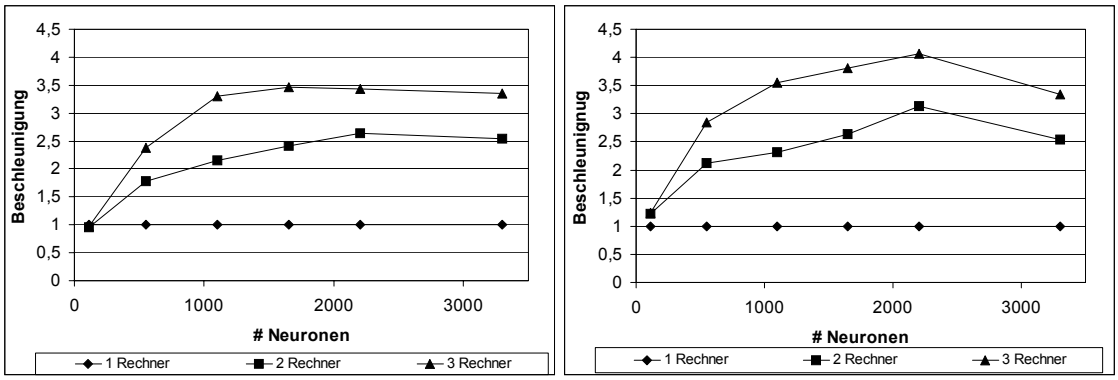


Abbildung 61: Beschleunigung der Simulation eines geschichteten Netzwerks[45] durch Verteilung der Simulation auf zwei und drei Rechnersysteme, links bezogen auf die schnellste Einzelplatzsimulation und rechts bezogen auf den Mittelwert der Einzelplatzsimulationen.

Abbildung 61 und Abbildung 62 zeigen die Simulationszeiten für das geschichtete Netzwerk bei einer Aktivität von 16 Pulsen pro Sekunde für ein bis fünf Rechner. Links ist die erzielte Beschleunigung im Vergleich zur Simulation auf dem schnellsten Rechner und rechts bezogen auf den Mittelwert aus allen an der Simulation beteiligten Rech-

[45] Netzwerk mit 11 Schichten und einer Aktivität von 16 Pulsen pro Sekunde ohne gebündelte Verarbeitung der Ereignisse

nern dargestellt. Bei den Messungen mit zwei und drei Rechnern diente Rechner A als Vergleichsmaßstab Bei den Messungen mit vier und fünf Rechnern wurde Rechner G zum Vergleich herangezogen.

Bereits ab einer Netzwerkgröße von ca. 881 Neuronen, also 80 Neuronen pro Schicht und 11 Schichten insgesamt, wird sowohl mit zwei als auch drei Rechnern eine Beschleunigung um den Faktor zwei, respektive drei erreicht. Wird der Mittelwert der Rechner als Bezugsgröße herangezogen, so gilt dies sogar schon ab 551 Neuronen, also einer Schichtgröße von 50 Neuronen. Der Vergleich mit dem Mittelwert ist nur bis zu einer Netzwerkgröße von 2201 Neuronen möglich, da die Simulation größerer Netze auf allen Rechnern, außer Rechner A, mangels entsprechenden Speicherausbaus nicht möglich war. Daher werden die Messungen mit 3301 Neuronen stets im Vergleich mit der Simulationsleistung des Rechners A dargestellt. Die Beschleunigung steigt für größere Netzwerke über die Zahl der verwendeten Rechner, da diese Netzwerkgrößen auf dem Einzelplatzrechner bereits an die Grenzen der dort verfügbaren Ressourcen stoßen bzw. diese überschreiten. Dies äußert sich zum Beispiel durch das Auslagern von Daten auf die Festplatte, wie dies in Abschnitt 4.3.2.2 am Beispiel von Simulationen mit der *calendar queue*, basierend auf unterschiedlichen Containerklassen, erläutert wurde.

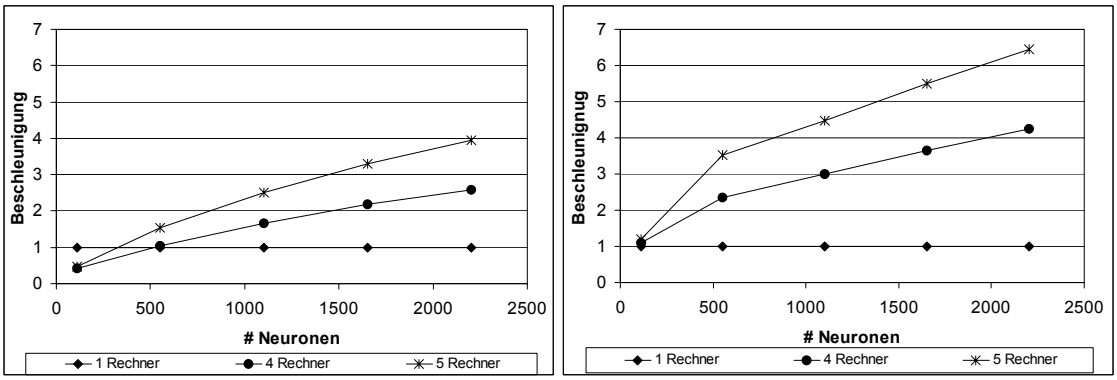


Abbildung 62: Beschleunigung der Simulation eines geschichteten Netzwerks durch Verteilung der Simulation auf vier und fünf Rechnersysteme, links bezogen auf die schnellste Einzelplatzsimulation und rechts bezogen auf den Mittelwert der Einzelplatzsimulationen.

Aus Abbildung 58 ist zu entnehmen, dass ohne gebündelte Verarbeitung die Simulationsleistung für die Einzelplatzsimulation nach einem kurzen Anstieg auf ein Maximum wieder abfällt und ab ca. 551 Neuronen, also einer Schichtbreite von 50 Neuronen, in einen konstanten Verlauf übergeht, der sich bis zu 1321 Neuronen (130 Neuronen Schichtbreite) erstreckt. Da dieser Verlauf mehr oder minder für alle an der Simulation beteiligten Rechner gilt, stellt sich ein der Rechneranzahl entsprechender Beschleunigungsfaktor in diesem Bereich ein. Während nach diesem Bereich für den Einzelplatzrechner die Simulationsleistung wieder abfällt, steigt sie in der verteilten Simulation weiter an, denn jeder einzelne Rechner hat weniger Neuronen zu simulieren. Erst oberhalb von 2201 Neuronen machen sich die Kommunikationskosten bemerkbar, so dass die Simulationsleistung der verteilten Simulation wieder etwas abnimmt.

Das vollvernetzte Netzwerk hat relativ zur Zahl der Neuronen eine deutlich höhere Zahl von Verbindungen als das geschichtete Netzwerk. Daher fällt in Vergleich zum geschichteten Netzwerk die maximale Netzwerkgröße geringer aus und und es tritt ein wesentlich höheres Kommunikationsaufkommen auf.

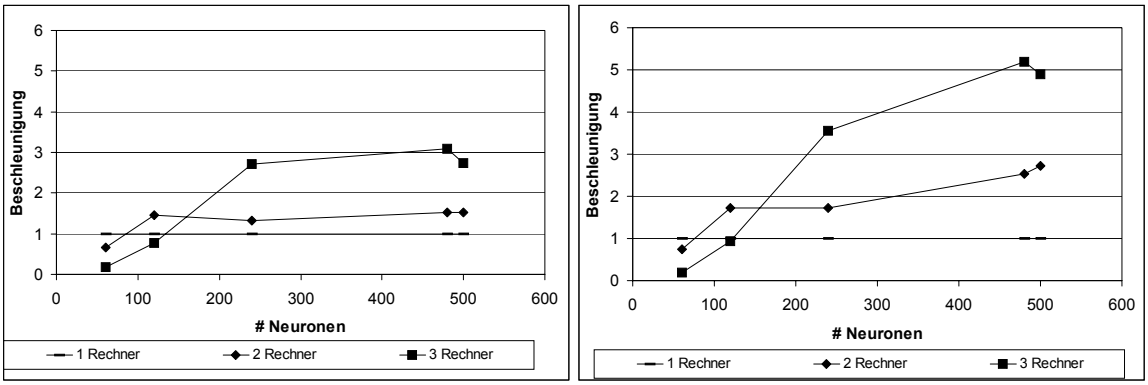


Abbildung 63: Beschleunigung der Simulation eines voll vernetzten Netzwerks[46] durch Verteilung der Simulation auf zwei und drei Rechnersysteme, links bezogen auf die schnellste Einzelplatzsimulation und rechts bezogen auf den Mittelwert der Einzelplatzsimulationen.

Bei der Simulation geschichteter Netzwerke wird ein Beschleunigungsfaktor in der Größenordung der Rechneranzahl mit Netzwerkgrößen erreicht, die der maximal simulierbaren Größe vollvernetzter Netzwerk entspricht. Werden diese vollvernetzten Netzwerke mit mehr als zwei Rechnern simuliert, so kann man höhere Beschleunigungsfaktoren beobachten als bei vergleichbar großen geschichteten Netzwerken.

Werden jedoch nur zwei Rechner verwendet, ergibt sich eine geringere Beschleunigung als beim geschichteten Netzwerk (1,5 zu 1,7). Dies kann darauf zurückgeführt werden, dass bei der Verwendung von lediglich zwei Rechnern die Auslastung der beteiligten Rechner nach wie vor sehr hoch ist und dass auf Grund des höheren Kommunikationsaufkommens eher Kollisionen während der Kommunikation entstehen.

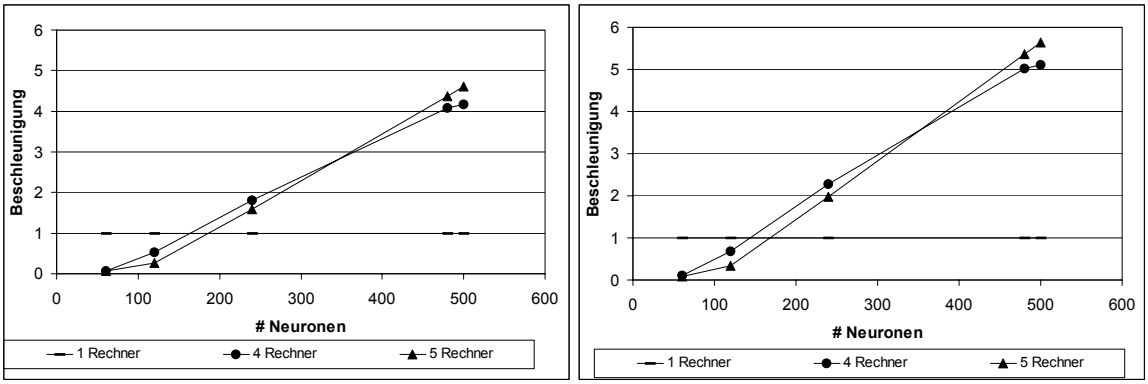


Abbildung 64: Beschleunigung der Simulation eines voll vernetzten Netzwerks[47] durch Verteilung der Simulation auf vier und fünf Rechnersysteme, rechts bezogen auf die schnellste Einzelplatzsimulation und links bezogen auf den Mittelwert der Einzelplatzsimulationen.

[46] Das Netzwerk wurde mit einer Aktivität von 16 Pulsen pro Sekunde ohne gebündelte Verarbeitung der Ereignisse simuliert.

Da die Verzögerungen zwischen den Neuronen eine um einen Mittelwert zufällige Größe besitzen, werden auch die betreffenden Nachrichtenpakete zu verschiedenen Zeitpunkten versendet. Je mehr Rechner in eine solche Simulation eingebunden werden, umso weniger wahrscheinlich überlappen sich die zeitlich zueinander verschobenen Nachrichtenpakete zweier Rechner. Hinzu kommt, dass ohnehin die Simulationsleistung für das vollvernetzte Netzwerk bei der Einzelplatzversion für größere Netzwerke rascher abfällt als bei dem geschichteten Netzwerk und daher insbesondere mehrere kleine Netzwerke merklich schneller simuliert werden als wenige große, solange eine ausreichend schnelle Kommunikation zur Verfügung steht.

4.3.3.3. Ergebnisse und Folgerungen

Wie die Messungen aus dem vorangegangenen Abschnitt zeigen, nimmt bei Verteilung der ereignisgetriebenen Simulation auf bis zu fünf Rechnern die Simulationsdauer stetigen ab.

Darüber hinaus ist es möglich, durch die Verteilung der Simulation Echtzeitbedingungen zu erfüllen. Abbildung 65 zeigt für ein geschichtetes Netzwerk mit 551 Neuronen, dass sich die Simulationsdauer bei Einsatz von vier Rechnern auf die zu simulierende Zeit von 10 Sekunden reduziert und mit fünf Rechnern noch weiter unterboten wird.

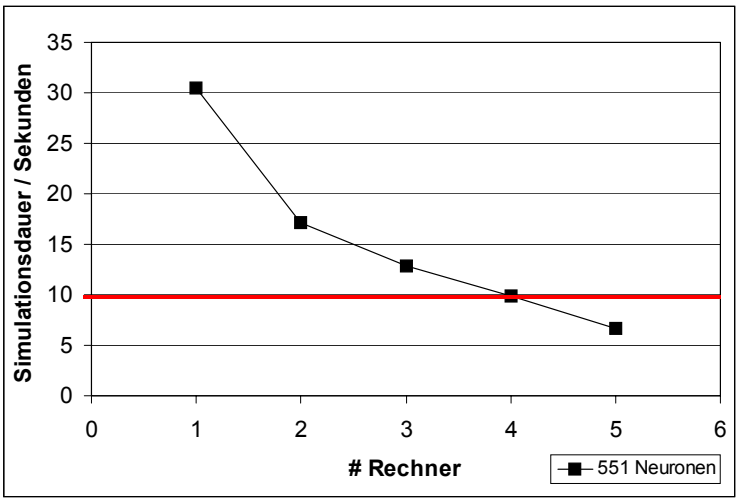


Abbildung 65: Simulation in Echtzeit mit Hilfe der verteilten Simulation. (551 Neuronen Netzwerk – Echtzeitbedingungen bei einer Verteilung auf vier Rechner) Simulierte Zeit: 10 Sekunden.

Eine verteilte Simulation interessiert vor allem aufgrund ihrer verkürzten Simulationsdauer. Sie ermöglicht es überdies, Netzwerkgrößen zu simulieren, die von einem einzelnen Rechner aufgrund seiner beschränkten Ressourcen nicht nur ineffizient, sondern gar nicht mehr bewältigt werden können. Aus dem vorangegangen Abschnitt geht hervor, dass ab einer Netzwerkgröße von 3301 Neuronen lediglich Rechner A ausreichende Ressourcen für eine Einzelplatzsimulation besaß. Gleiches gilt für die Simulation des voll vernetzten Netzwerks mit 600 Neuronen, deren Simulationszeiten für unterschiedliche Verteilungen in Abbildung 66 dargestellt sind. Der raschere Abfall der Simulationszeit beim voll vernetzten Netzwerk auf einen asymptotischen Wert ist auf die

[47] Das Netzwerk wurde mit einer Aktivität von 16 Pulsen pro Sekunde ohne gebündelte Verarbeitung der Ereignisse simuliert.

Leistungscharakteristik der verwendeten Messrechner zurückzuführen, da Rechner A, der aufgrund des größten Speicherausbaus zum Vergleich herangezogen werden muss, durch den kleineren Cache gerade beim voll vernetzten Netzwerk eine schlechtere Performance aufweist. Dieser Effekt tritt dagegen im geschichteten Netzwerk nicht auf (siehe auch Abbildung 52 und Abbildung 53).

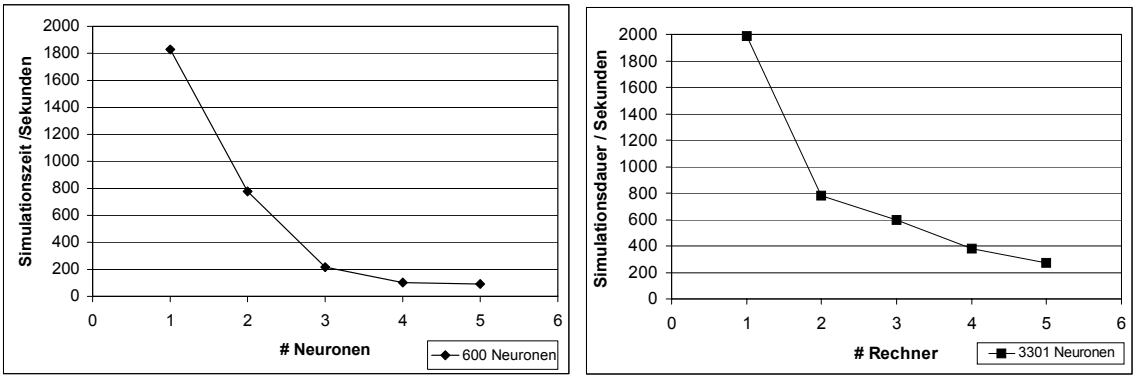


Abbildung 66: Simulation eines voll vernetzten Netzwerks mit 600 Neuronen und eines geschichteten Netzwerks mit 3301 Neuronen auf ein bis fünf Rechner.

Daher ist es möglich, durch die Verteilung der Simulation weitaus größere Netzwerke in vertretbarer Zeit zu simulieren als mit einer Einzelplatzsimulation. Dies gilt selbst dann noch, wenn eine maximale Kommunikation, wie bei dem voll vernetzten Netzwerk, zwischen den beteiligten Rechnern stattfindet.

Kapitel 5 - Hardware für die Simulation pulsverarbeitender neuronaler Netze

Eine vollständige Abbildung pulsverarbeitender neuronaler Netze in Hardware gestaltet sich schwierig, da die Simulation pulsverarbeitender neuronaler Netze eine große Flexibilität in der Wahl der Modellparameter fordern. Dies ist dadurch bedingt, dass einige Eigenschaften pulsverarbeitender neuronaler Netze nach wie vor Gegenstand der Forschung sind und aus diesem Grunde von vornherein für eine Simulation flexibel gehalten werden müssen. Eine effiziente Abbildung in Hardware fordert jedoch in der Regel Einschränkungen entweder in der Flexibilität der Parameterwahl, der Berechnungsgenauigkeit oder der Vernetzungsstruktur. Denn nur durch die Konzentration auf wesentliche funktionale Aspekte kann eine Hardware-Implementierung einen deutlichen Gewinn gegenüber einer Softwaresimulation bieten. Dennoch bietet es sich an, zumindest Teilaspekte der Simulation in Hardware zu beschleunigen, um einerseits die Simulation besonders großer Netze in vertretbarer Zeit zu gewährleisten, andererseits um für Anwendungen dieser Netzwerke Simulationen in Echtzeit zu ermöglichen. Werden die Funktionseinheiten mittels programmierbarer Logik in Hardware abgebildet, kann darüber hinaus mit Hilfe der bereits bestehenden VHDL[48]-Beschreibungen schneller in einen hoch integrierten Schaltkreis migriert werden. Auch eine Anpassung aufgrund neuer Forschungsergebnisse wird dadurch erleichtert bzw. erst möglich.

Auf Hardware-Implementierungen pulsverarbeitender neuronaler Netze in Analogtechnik, die auf eine möglichst getreue Abbildungen einzelner neuronaler Funktionen und auf Eigenschaften wie einen geringen Leistungsbedarf abzielen, wird hier nicht näher eingegangen, da bei diesen Implementierungen bisher nur wenige Neuronen modelliert werden können.

5.1. Beschleunigung der ereignisgetriebenen und verteilten Simulation pulsverarbeitender neuronaler Netze

Eine Beschleunigung der vorgestellten ereignisgetriebenen und verteilten Simulation scheint mit Rücksicht auf die Messungen aus Abschnitt 4.3.2 und 4.3.3 für die Kommunikation, Ereignisliste und Neuronenmodelle angebracht.

5.1.1. Beschleunigung der Kommunikation

Die Effizienz einer verteilten Simulation ist vor allem von der Latenz und der Bandbreite der zugrunde liegenden Kommunikation abhängig. Wünschenswert wäre eine Hardwareeinheit, die beide Bedingungen erfüllt, wobei sie die Eigenschaften sowohl der Simulation als auch der die Information tragenden Ereignisse für Optimierungen

[48] **V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage

nutzen sollte. Durch eine möglichst gute Anpassung an die Bedürfnisse der Simulation kann vermieden werden, dass die zu transferierenden Daten aufwendig aufbereitet werden müssen. Dadurch kann eine bessere Entlastung des Prozessors erreicht werden als bei der Verwendung von Standardprotokollen und -hardware.

5.1.2. Beschleunigung der Ereignisliste

Die Ereignisliste muss in unregelmäßigen Abständen große Ereignismengen aufnehmen und sortieren. Gelingt es, die Sortierung der Ereignisse von dem eigentlichen Simulationsprozess zu entkoppeln, so kann durch eine in Hardware realisierte Ereignisliste der Prozessor von dieser Aufgabe entlastet werden. Abhängig vom gewählten Algorithmus kann die Verwaltung der Ereignisliste einen wesentlichen Anteil der Rechenzeit in Anspruch nehmen. In einem solchen Falle verspricht eine Hardwarebeschleunigung einen Gewinn. Die hohe Frequenz, mit der auf die Ereignisliste zugegriffen wird, erschwert jedoch die notwendige Entkopplung von dem eigentlichen Simulationsprozess. Darüber hinaus ist der zu erwartende Gewinn sehr von der Netzwerkstruktur und der Netzwerkaktivität abhängig.

5.1.3. Beschleunigung der Neuronenmodelle

Je aufwendiger die Berechnung eines Neuronenmodells ist, desto eher bietet sich die Beschleunigung eines solchen Modells in Hardware an, wobei sich die zugrunde liegenden Algorithmen für eine Abbildung in Hardware eigenen müssen. Gelingt es, mehrere Neuronenmodelle parallel in Hardware zu beschleunigen oder zumindest die Parameter mehrerer Neuronenmodelle in der Hardware bereitzuhalten und eine Verarbeitung in einer Pipeline zu realisieren, so entsteht eine Art Subsimulator, wie er in der verteilten Simulation existiert, mit dem Unterschied, dass durch bessere Anbindung der beschleunigenden Hardware ein effizienterer Datenaustausch möglich wird.

Bei der Beschleunigung einzelner Neuronen ist von entscheidender Bedeutung, ob in dem zu simulierenden Netzwerk häufig Neuronen unabhängig voneinander berechnet werden können, da sie sich nicht direkt gegenseitig beeinflussen, und ob mittelbare Einflüsse außerhalb des sicheren Zeitfensters für die Berechnung liegen. Gelingt es hingegen, mehrere Neuronen in Hardware zu beschleunigen, so gelten für einen Leistungsgewinn im weitesten Sinne die gleichen Randbedingungen wie bei der verteilten Simulation.

5.2. Neurocomputer und Simulationsbeschleuniger

Es gibt wenige Beispiele digitaler Hardware, die speziell für die Simulation pulsverarbeitender neuronaler Netze entwickelt wurden: *Spike128K* und *ParSpike* aus der Universität Paderborn sowie *NESPINN* und *MASPINN* aus der Technischen Universität Berlin. Die theoretischen Ansätze für diese Architekturen gehen auf die Arbeiten von Eckhorn und Reitböck et.al. [EckBauJor1988], [EckReiArn1989] zurück.

Auf die vielen verschiedenen Hardware-Implementierungen, die der beschleunigten Berechnung ratencodierter neuronaler Netze dienen, wie zum Beispiel SYNAPSE1 [RamBeiBrü1991], wird hier nicht weiter eingegangen. Einen guten Überblick über solche Systeme bietet [Zell1994].

Neben den digitalen Hardwarebeschleunigern für neuronale Netze, den so genannten Neurocomputern, existieren digitale Simulationsbeschleuniger, welche allgemein auf

die Beschleunigung einer ereignisgetriebenen Simulation ausgelegt sind. Für eine Übersicht über diese Systeme sei auf [Luksch1993] verwiesen.

5.2.1. Neurocomputer - digitale Hardwarebeschleunigung für PVNN

Bei den Systemen[49] *Spike128K*, *ParSpike*, *NESPINN* und *MASPINN* handelt es sich um Zeitscheibensimulationen, die durch den Einsatz von Ereignislisten optimiert wurden. Aktive, feuernde Neuronen werden in einer Spikeliste gehalten, und nur für diese Neuronen und Neuronen, die ein inneres Potential über einer fest definierten Schwelle nahe Null besitzen, findet eine Berechnung statt. Da die Simulation nach dem Zeitscheibenverfahren arbeitet, reicht es, die von den Neuronen ausgehenden Pulse zu speichern. Liegt kein Puls am Eingang eines Neurons an, so wird der Zerfall des inneren Potentials berechnet (Abklingphase), bis dieses unter die eingangs erwähnte Schwelle fällt. Allen Systemen gemeinsam ist die Festlegung auf eine Zeitscheibe von 1 *ms* und weitestgehend eine Optimierung auf das Eckhorn-Neuronenmodell (siehe Abschnitt 2.2.5). Außerdem ist die Modellierung von Verzögerungen in diesen Systemen, wenn überhaupt, nur in sehr rudimentärer Weise möglich, da Zwischenspeicher geringer Tiefe eine Verzögerung der Ausgangspulse um maximal 8 Zeitschritte ermöglichen. Individuelle Verzögerungen für einzelne Verbindung zwischen einem Ausgang des einen Neurons und einem Eingang eines anderen Neurons sind grundsätzlich nicht möglich.

5.2.1.1. Spike128K

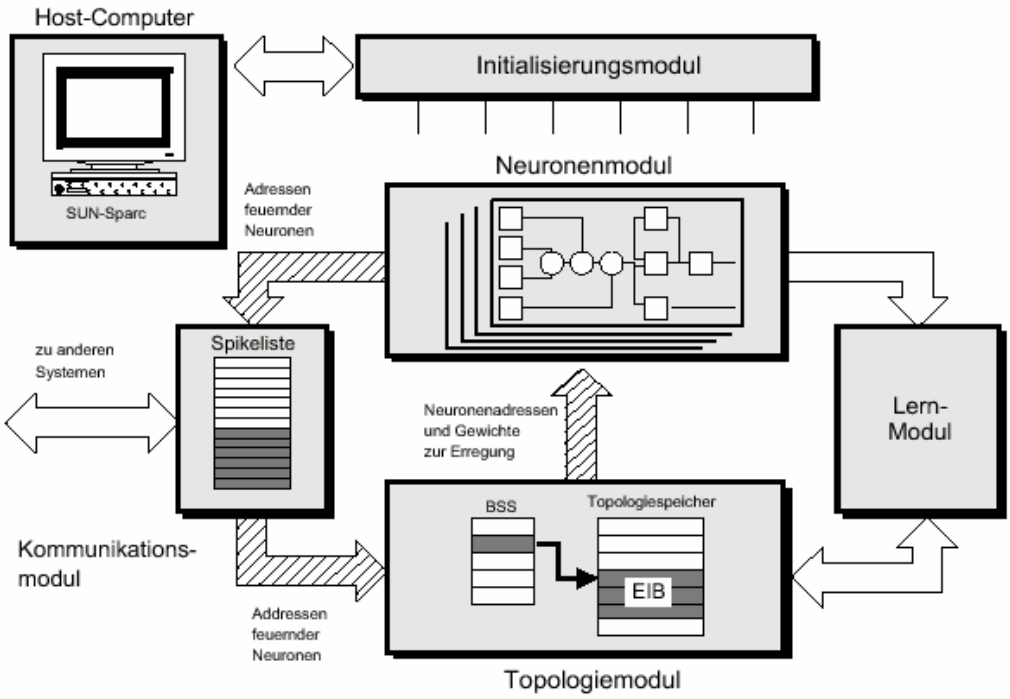


Abbildung 67: Blockschaltbild des Spike128k Systems aus [Wolff2001]

[49] Die Darstellungen der Systeme Spike128K und ParSpike basieren auf [HarFraSch1997] und [Wolff2001]. Die Darstellungen der Systeme NESPINN und MASPINN auf [JahRotKla1996], [SchMehJah1998], [SchMehKla1998] und [SchAtaMeh2000].

SPIKE128K bildet den Simulationsalgorithmus in Hardware ab, indem die Berechnungsschritte für ein Neuron bzw. eine Synapse in Form einer Verarbeitungspipeline aufgebaut werden. Die Abklingphase wird vom Neuronenmodul behandelt, die Erregungsphase übernimmt das Topologiemodul. Die Module sind über eine Spikeliste repräsentiert durch das Kommunikationsmodul miteinander gekoppelt, welches die ausgehenden Pulse der Neuronen speichert. Zur Adaption der Gewichte wurde zudem durch die Arbeiten [SchHar1999][Schäfer2000] ein Lernmodul beigestellt. Ein VME-Bus-System integriert die Module zum Spike128K-System.

Die Simulationsgeschwindigkeit wird beim SPIKE128K durch den Einsatz von Pipelining und Parallelität gesteigert. Sequentielle Anteile des Simulationsalgorithmus, die für ein Pipelining genutzt werden, sind zum Beispiel die Berechnung des Membranpotentials und der Vergleich mit der dynamischen Schwelle. Parallelität wird durch das gleichzeitige Abklingen aller Teilpotentiale eines Neurons erreicht, da für jedes dieser Teilpotentiale eine eigene Arithmetikeinheit vorhanden ist. SPIKE128K ist als Prototypsystem auf Basis programmierbarer Logik und kommerzieller Arithmetikeinheiten und DRAM bzw. SRAM-Speichern aufgebaut. Der Pipelinetakt des Systems beträgt 10 MHz. Die theoretische Simulationsgeschwindigkeit für einen Zeitschritt ist laut [Wolff2001] 4,5*2 ms*. Theoretisch findet die Behandlung der Spikeliste durch das Kommunikationsmodul parallel zu den Berechnungen statt und kann als Vor- oder Nachfolgestufe der jeweiligen Berechnungspipeline angesehen werden. Da eine optimale Überlappung der Stufen in Spike128K nicht erreicht wird, bleibt die theoretische Geschwindigkeit unerreicht. Die gemessene Geschwindigkeit liegt bei 11 ms pro Zeitschritt (1ms).

Das SPIKE128K System kann 128.000 Neuronen mit insgesamt 16 Millionen Verbindungen verwalten und ist an ein Sun-Hostsystem über einen Transputerlink mit der theoretischen Datenrate von 20 Mbit/s angebunden. Durch aufwendige in Software realisierte Protokolle und die geringe Rechenleistung der Transputer entsteht damit eine Latenzzeit für jedes Datenpaket, welche sich insbesondere bei einer hohen Anzahl kleiner Datenpakete auf die Gesamtsimulationsdauer niederschlägt.

5.2.1.2. ParSpike

Bei ParSPIKE handelt es sich um ein Systemkonzept, welches nicht implementiert wurde. Es lehnt sich in seiner Architektur an das Spike128K System an, setzt jedoch digitale Signalprozessoren (DSPs) als Rechenknoten ein.

Auch in diesem System soll eine SUN Workstation als Host eingesetzt werden. Ein VME-Bus dient zur Kopplung so genannter nrc- bzw. rc-Module[50]. Die Module tragen neben den DSPs als Berechnungsknoten zusätzlich Kommunikationsmodule, welche die Verwaltung und Übermittlung der erzeugten Pulse bewerkstelligen.

Das nrc-Modul dient dazu, nicht reguläre Verbindungsstrukturen mit bis zu 256.000 Neuronen zu unterstützen, und hält dafür einen globalen Topologiespeicher bereit. Das rc-Modul kann hingegen 512.000 Neuronen simulieren, die jedoch nur über eine reguläre Netzwerkstruktur verknüpft werden können, da anstelle des globalen Topologiespeichers Verknüpfungsmasken in den lokalen Speichern eines jeden Rechenknotens zum Einsatz kommen.

[50] nrc = non regular connections, rc = regular connections

Ein wesentlicher Vorteil dieses Konzepts ist, dass es sich leichter programmieren lässt, da es auf Standardsignalprozessoren basiert und lediglich für die kritische Kommunikation Spezialhardware vorhält. Eine Abschätzung der Simulationsleistung findet sich am Ende des Abschnitts 5.2.1.

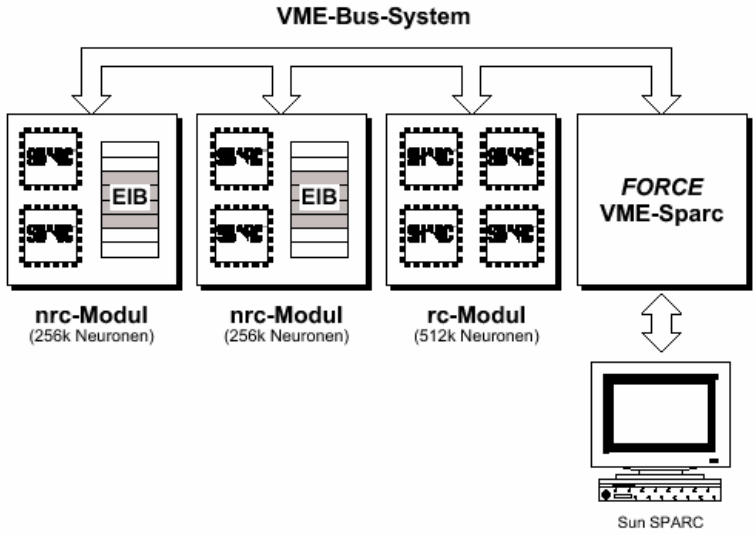


Abbildung 68: ParSPIKE Systemkonzept (übernommen aus [Wolff2001]).

5.2.1.3. NESPINN

Bei NESPINN handelt es sich um ein Konzept für einen Neuroprozessor, welcher zwar schaltungstechnisch entworfen, nicht jedoch gefertigt wurde.

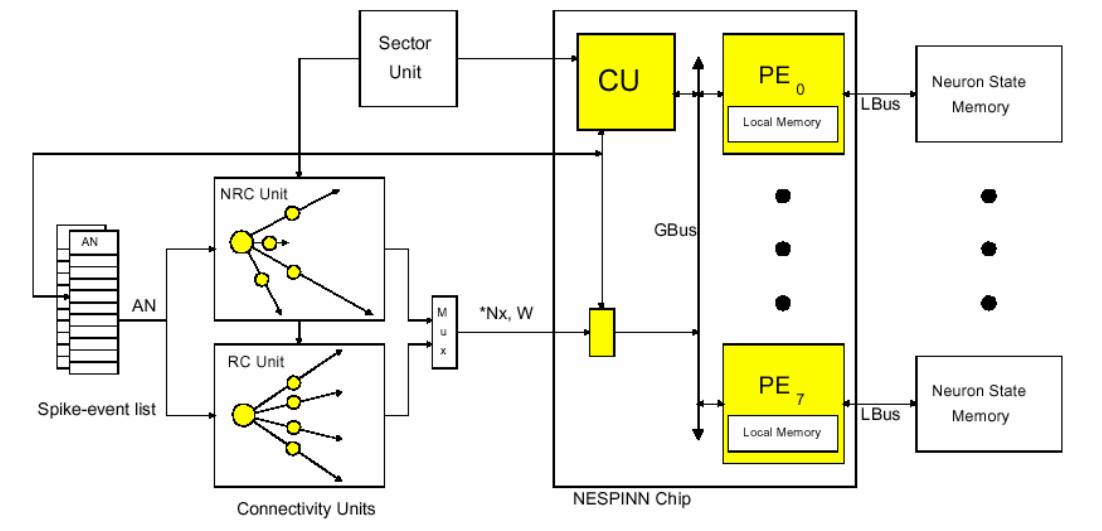


Abbildung 69: System mit NESPINN Chip (übernommen aus [JahRotKla1996])

Vereinfacht kann NESPINN als SoC[51]-Variante des Spike128K Systems aufgefasst werden, wobei jedoch einerseits das Design weiterentwickelt wurde und andererseits verschiedene Aspekte des Systems anders umgesetzt werden mussten. Der NESPINN-Prozessor wurde mit einer Taktfrequenz von 50 MHz ausgelegt und verarbeitet

[51] SoC = **S**ystem **o**n a **C**hip

dendritische Potentiale in vier Prozessorpipelines. Über eine Markierung (Tags) der Potentiale wird eine Beschränkung auf aktive Potentiale erreicht. Die Neuronendaten sind in externen SRAM-Speichern abgelegt. Für die Topologieverarbeitung und die Spikelistenverwaltung sind ebenfalls externe Einheiten vorgesehen.

5.2.1.4. MASPINN

Das MASPINN-System [SchMehJah1998][SchMehKla1998] ist eine Weiterentwicklung des NESPINN-Konzepts. Es wurde dabei ein besonderes Augenmerk auf eine optimierte Nutzung des externen SRAM-Speichers gelegt, um die zwei Pipelines des NeuroPipe-Chips möglichst kontinuierlich mit Daten speisen zu können. Um dies zu erreichen, werden die Abkling- und Erregungsphasen miteinander verschränkt, da die Akkumulation der Potentialmodifikationen durch die Verbindungsgewichte mittels einer externen Topologiedateneinheit erfolgt. Zudem existiert zusätzlich ein TAG-RAM, mit Hilfe dessen die aktiven zu verarbeitenden Potentiale verwaltet werden.

Von dem gesamten System wurde der NeuroPipe-Chip realisiert und simuliert [SchAtaMeh2000]. Da die externe Topologieeinheit bisher nicht realisiert wurde, sind die weiter unten angegebenen Simulationsgeschwindigkeiten für die MASPINN-Berechnungen nur Schätzungen. Die Taktfrequenz des NeuroPipe-Chips beträgt 100 MHz.

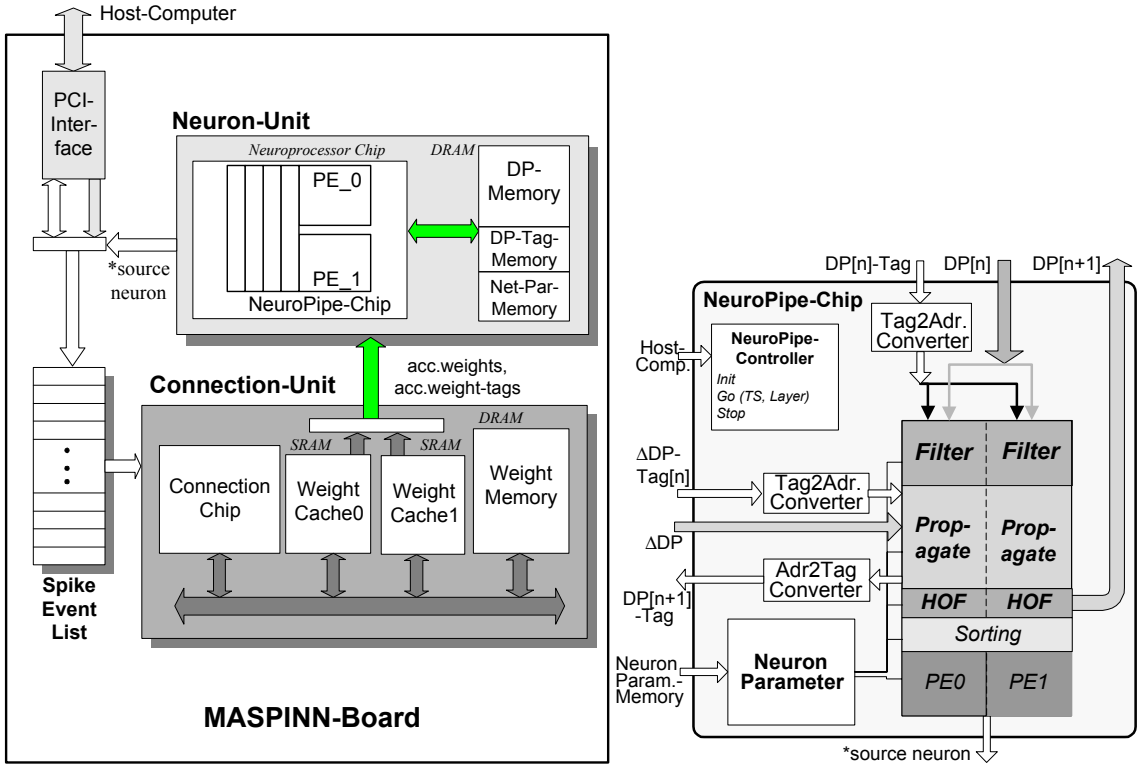


Abbildung 70: Blockschaltbilder des MASPINN-Systems und des darin enthaltenen NeuroPipe-Chips (aus [GraSchWol2002])

5.2.1.5. Vergleich der Simulationsleistungen

Aus [Wolff2001] stammt die folgende Tabelle, welche die Simulationsleistungen der vorgestellten Systeme vergleicht. Für die Systeme, welche nicht implementiert wurden, sind die Werte aufgrund der architekturellen Eigenschaften abgeschätzt worden.

Anzahl der Neuronen	**128.000**	**512.000**	**1.000.000**
MASPINN (2 PE/100 MHz)	0,8 ms	3,2 ms	6,4 ms
NESPINN (4 PE/50 MHz)	3 ms	11 ms	23,4 ms
SPIKE128k (1-8 PE/10 MHz)	10 ms	11 ms (4PE)	12 ms (8PE)
ParSPIKE (64 PE/AD21060/40 MHz)	2,5 ms	10 ms	20 ms
ParSPIKE (64 PE/AD21160/100 MHz)	1 ms	4 ms	8 ms
ParSPIKE (64 PE/TS001/150 MHz)	0,7 ms	2,7 ms	5,4 ms
ParSPIKE (64 PE/TSxxx/250 MHz)	0,4 ms	1,6 ms	3,2 ms

Tabelle 9: Vergleich der Simulationsgeschwindigkeiten (übernommen aus [Wolff2001])[52].

[52] ADSP21060, ADSP21160, TigerSHARC TS001, TigerSHARC TSxxx sind unterschiedliche DSPs die als Rechnerknoten fungieren (siehe [Wolff2001]).

Kapitel 6 - RACER - Hardware zur Beschleunigung der Simulation und Integration von pulserzeugender Hardware

Die zuvor beschriebenen Beschleuniger und Spezialprozessoren wurden jeweils auf ein bestimmtes Neuronenmodell hin optimiert. Solche Einschränkungen versprechen einen spürbaren Leistungsgewinn durch den Einsatz von Spezialhardware. Eine solches Vorgehen erscheint im Falle pulsverarbeitender neuronaler Netze gewagt, da die zugrunde liegenden Modelle noch Gegenstand der Forschung sind. Neue Erkenntnisse darüber, wie Informationen in solchen Netzwerken verarbeitet werden, vor allem aber Erkenntnisse über die wesentlichen funktionalen Aspekte der Neuronmodelle, können entweder nur sehr eingeschränkt oder gar nicht durch diese Architekturen abgebildet werden.

Die RACER-Hardware setzt deshalb andere Schwerpunkte. Sie ermöglicht programmierbare Verarbeitungseinheiten und stellt eine möglichst einfache und effiziente Kommunikation zum Zugriff auf diese Einheiten bereit. Die gesteigerte Flexibilität wird durch geringere Leistungsfähigkeit erkauft, da programmierbare Schaltkreise nicht die Leistungsfähigkeit von Spezialprozessoren erreichen können. Ziel dieser Entwicklung ist eine hochflexible Entwicklungsplattform, die es erlaubt, verschiedenste Ansätze zur Beschleunigung von Teilaufgaben der Simulation in Hardware zu evaluieren und die Entwicklung von Vorverarbeitungsstufen zu unterstützen, die entweder analoge Signale verarbeiten oder dazu dienen, pulserzeugende analoge oder digitale Hardware in die Simulation zu integrieren.

Um diesen Anforderungen gerecht zu werden, wurde eine Datenflussarchitektur implementiert, welche drei weitgehend unabhängige Datenströme über frei wählbare Verarbeitungsmodule verfügbar macht.

6.1. Aufbau der Evaluierungsplattform (PLD[53]-Karte)

6.1.1. Architektur

Abbildung 71 zeigt das Blockschaltbild der Beschleunigerkarte. Drei kaskadierbare Module werden über einen gemeinsamen Bus mit Daten beschickt und speisen die berechneten Daten über denselben wieder ins System ein. Die Ankopplung an den PCI-Bus erfolgt über einen Controller der Firma AMCC.

Die Steuerung der Erweiterungskarte und die Kontrolle des Interfaces zum PCI-Controller erfolgt durch einen komplexen FPGA[54], den so genannten lokalen Controller. Ein

[53] **P**rogrammable **L**ogical **D**evice

[54] **F**ield **P**rogrammable **G**ate **A**rray

weiterer, weniger komplexer FPGA dient dazu, die auf den Modulen befindlichen programmierbaren Bausteine zu programmieren.

Große FIFOs (512 kB) entkoppeln sowohl bei der Programmier- und Testeinheit (PTU[55]) eingangsseitig wie auch bei jedes der Module eingangs- und ausgangsseitig. Diese Entkopplung ermöglicht bei den Modulen einen parallelen Betrieb. Bei der Programmierung mit der PTU stellt der FIFO den kontinuierlichen Strom der Programmierdaten sicher.

Mit Hilfe von Crossbar-Switches, welche über den lokalen Controller gesteuert werden, ist es möglich, die Module zu kaskadieren. Es lassen sich die Ausgangsdaten des ersten Moduls in das zweite und die Ausgangsdaten des zweiten Moduls in das dritte Modul einspeisen. Jeder der Switches lässt sich getrennt vom anderen steuern. Darüber hinaus ließe sich die Programmierung des lokalen Controllers derart modifizieren, dass die Ausgangsdaten des dritten Moduls auch wieder in das erste Modul eingespeist werden könnten und somit eine Zirkulation der Daten möglich wäre.

Die PTU unterstützt bis zu zwei programmierbare Bausteine auf jedem Modul, wobei sich auch nicht programmierbare Funktionseinheiten auf den Modulen befinden können.

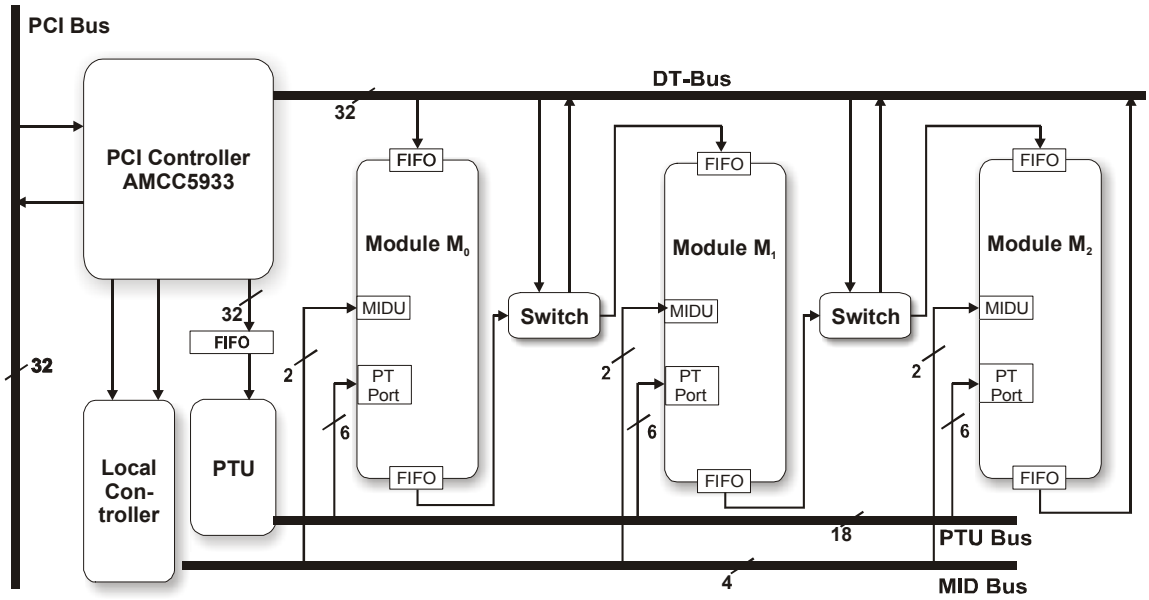


Abbildung 71: Blockschaltbild der RACER – Beschleunigerkarte.

6.1.2. Die Realisierung

Bei der RACER Beschleunigerkarte handelt es sich um eine sechslagige, lange PCI Erweiterungskarte. Abbildung 72 zeigt die Erweiterungskarte bestückt mit einem Modul auf dem ersten Modulsteckplatz. Zwischen den Steckverbindern der unbelegten Modulsteckplätze sind die beiden FIFO Bausteine für den Modulsteckplatz wie auch die Crossbar-Switches zu erkennen. Links auf der Erweiterungskarte, oberhalb des PCI-Steckverbinders, befinden sich oberhalb des AMCC PCI-Controllers der FPGA des lokalen Controllers und links oberhalb der FPGA der Programmier- und Testeinheit. Alle Verbindungen zwischen den PCI-Controller und dem lokalen Controller sind über so genannte zero-delay-Switches miteinander gekoppelt, da der PCI-Controller mit 5 Volt

[55] **P**rogramming and **T**est **U**nit

arbeitet und alle anderen Schaltkreise der Erweiterungskarte mit 3,3 Volt arbeiten. Am linken Rand der Karte sind drei EEPROMs zu erkennen, wovon zwei die Programmierung des lokalen Controllers aufnehmen und einer die Programmierung der PTU. Um einen sicheren Betrieb der Module auf der Erweiterungskarte zu gewährleisten und gleichzeitig die PCI-Spezifikation von maximal 25 Watt pro Erweiterungskarte einzuhalten, wurde in der linken oberen Ecke der Erweiterungskarte eine zusätzliche Buchse zur Spannungsversorgung angebracht. So kann jedes der Module wahlweise über den PCI-Bus oder über diese Buchse mit Spannung versorgt werden.

An den Modulsockeln liegen 5 Volt, 3,3 Volt und eine zwischen 1 und 5 Volt über steckbare Widerstände eines Präzisionsspannungsreglers frei einstellbare Spannung an. Die Module werden überdies mit einem gepufferten PCI-Takt versorgt.

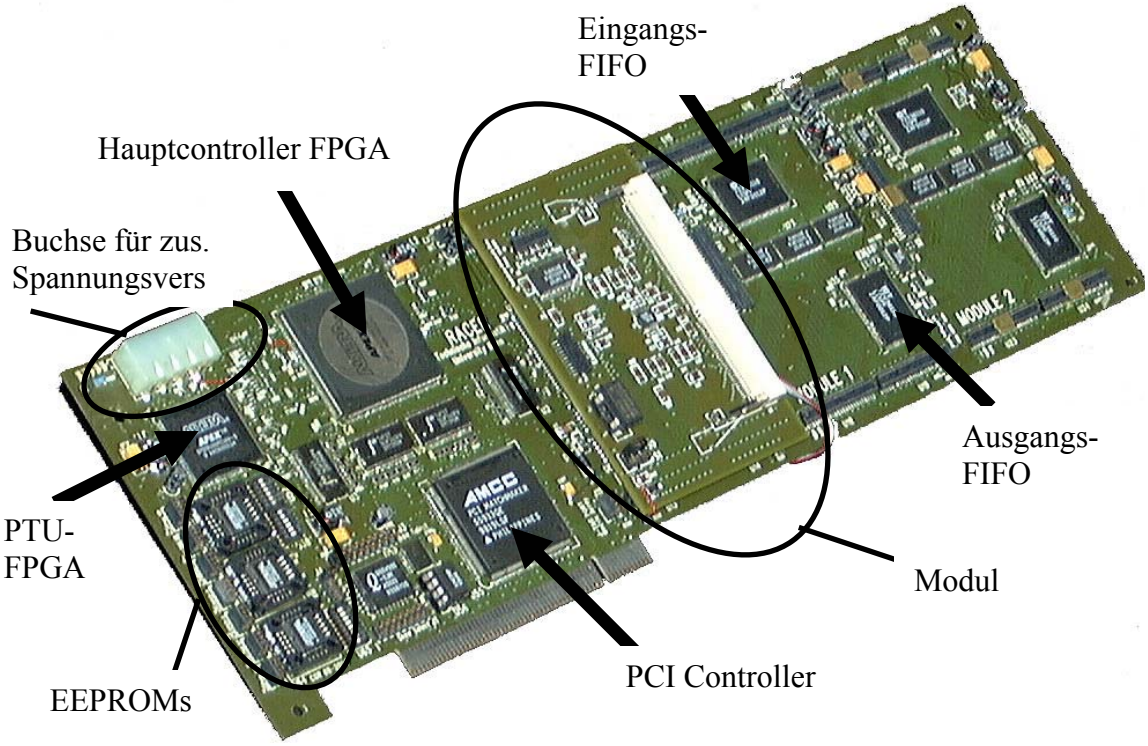


Abbildung 72: RACER PCI – Erweiterungskarte, bestückt mit einem Modul auf dem ersten Modulsteckplatz.

Das erste im Rahmen dieser Arbeit aufgebaute Modul für den RACER-Beschleuniger ist mit einem Altera Flex 10K200S FPGA, einem gesockelten TTL-Taktgeber und einer PLL[56] ausgestattet. Die PLL dient dazu, einen höheren synchronen Takt für den auf der Oberseite des Moduls befindlichen Speichersockel bereitzustellen. Der Sockel nimmt SO-DIMM Speichermodule mit einer Kapazität bis 256 MB auf. Das so ausgestattete Modul kann größere Datenmengen aufnehmen und dadurch längere Phasen autarker Berechnungen ermöglichen.

Neben 8 festen Eingängen, die über einen Stecker auf der Oberseite des Moduls an den FPGA herangeführt werden, können mit Hilfe eines vom FPGA steuerbaren Crossbar-Switch statt der oberen 12 Bit des 32 Bit FIFO-Eingangssignals 12 Signale eines zweiten Steckers von der Oberseite des Moduls an den FPGA geführt werden. Hierdurch ist

[56] **P**hase **L**ock **L**oop

es möglich, bis zu 20 digitale Signale von einer externen Quelle in den FPGA einzuspeisen, wobei gleichzeitig über die verbleibenden 20 Bit des FIFO-Eingangssignals Befehlssequenzen und Daten vom Host empfangen werden können.

In Abbildung 72 ist deutlich der unbestückte SO-DIMM Sockel auf der Oberseite des Moduls zu erkennen. Der FPGA befindet sich auf der Unterseite des Moduls. Eine detaillierte Beschreibung des Modulaufbaus, inklusive der Steckerbelegungen und Konfigurationsmöglichkeiten, findet sich in [Flieck2000].

Befehle an den lokalen Controller werden mit Hilfe der Mailboxen des AMCC-Controllers übermittelt. Wird eine der Mailboxen des AMCC-Controllers beschrieben, so löst dies auf Seiten der Erweiterungskarte einen Interrupt aus, welcher durch die Implementierung des lokalen Controllers behandelt wird.

Datentransfers in den FIFO der Programmier- und Testeinheit wie auch in die eingangsseitigen FIFOs der Module erfolgen mit Hilfe eines so genannten Pass-Thru Transfers. Tritt ein solcher Transfer auf, so wird dies vom AMCC-Controller über einen dedizierten Pin signalisiert. Die Implementierung des lokalen Controllers sorgt dann für die rechtzeitige Aktivierung der entsprechenden Steuersignale für die FIFOs.

Die von den Modulen berechneten Daten werden aus den ausgangsseitigen FIFOs der Module mit Hilfe eines Mastertransfers in den Speicher des Hosts übertragen. Die Auslösung eines solchen Transfers geschieht aufgrund eines der FIFO Füllstandssignale (leer (E), nahezu leer (AE), halb voll (HF), nahezu voll (AF) und voll (F))[57]. Welches dieser Signale den Transfer auslöst, lässt sich im lokalen Controller während des Betriebs programmieren. Der lokale Controller stellt zudem sicher, dass bei mehreren aktiven Auslösungssignalen die Module reihum die Gelegenheit erhalten, ihre Daten zu transferieren.

Neben den geschilderten Datentransfers stellt der lokale Controller beim Systemstart eine ordnungsgemäße Konfiguration des AMCC-Controllers und aller anderen Baugruppen auf der Erweiterungskarte sicher. Die Konfiguration der Erweiterungskarte und statistische Daten über die vergangenen Datentransfers lassen sich über einen entsprechenden Befehl vom lokalen Controller abrufen. Der lokale Controller überträgt in diesem Falle die Daten mit Hilfe eines Mastertransfers in einen eigens hierfür angelegten Speicherbereich des Hosts.

Für eine Programmierung mit Hilfe der Programmier- und Testeinheit sind lediglich entsprechende Daten in den FIFO der Programmier- und Testeinheit einzuspeisen. Die Daten müssen mit einem Prolog versehen sein, der angibt, welcher Baustein programmiert werden und mit welcher Taktrate dies geschehen soll. Sobald entsprechende Daten in den FIFO der Programmier- und Testeinheit eintreffen, beginnt diese, den betreffenden Baustein zu programmieren, sofern der Baustein zurzeit keine Berechnungen durchführt.

Die Dauer der Konfiguration hängt von der Größe des zu programmierenden Bausteins und von der maximalen Taktrate ab, mit der er programmiert werden kann. Bei der RACER-Hardware steht mit dem gepufferten PCI-Takt eine maximale Taktrate von 33 MHz zur Verfügung. Aktuelle Bausteine lassen sich mit Taktraten bis zu 33 MHz bzw. 57 MHz programmieren. Der Programmiertakt des Flex10K200S, welcher auf dem im Rahmen dieser Arbeit gefertigten Modul verbaut ist, kann maximal 16,7 MHz

[57] E - empty, AE - almost empty, HF – half full, AF – almost full und F - full

betragen. Die erforderliche Datenmenge für eine Konfiguration des Bausteins beträgt 2.740.000 Bits. Daraus ergibt sich eine Konfigurationsdauer von 164,4 ms für diesen Baustein. Im Vergleich dazu benötigt die Konfiguration des zurzeit größten von der Firma Altera verfügbaren Bausteins 360,33 ms, mit 12.011.000 Bits und einer Taktrate von 33 MHz. Die Rekonfiguration eines Moduls während des Betriebs lohnt sich daher nur, wenn die zwischen zwei Konfigurationsvorgängen stattfindenden Berechungen ein Vielfaches der angegebenen Konfigurationszeiten in Anspruch nehmen, es sei denn, beschränkte Ressourcen erfordern ohnehin eine Rekonfiguration.

6.1.3. Programmierumgebung

In [Sauermann2001] wird die Implementierung des RACER-Gerätetreibers für das Betriebssystem Microsoft Windows NT 4.0 beschrieben. An dieser Stelle wird die logische Sicht wiedergegeben, welche die bereitgestellte Software vermittelt, und die wichtigsten Eckdaten und funktionalen Aspekte der Implementierung werden erläutert. Durch Ergänzungen, die nach Fertigstellung der Software durch [Sauermann2001] erfolgten, weicht die vorliegende Darstellung in manchen Punkten von [Sauermann2001] ab.

6.1.3.1. Programmiermodell – logische Sicht

Der Zugriff auf die RACER-Hardware erfolgt durch gezielte Datentransfers auf die bereitstehenden Module. Daher wird von der Programmierumgebung das Senden (*send*) und Empfangen (*receive*) von Daten auf einzelne Module oder auf Kombinationen von mehreren Modulen unterstützt.

Dem Anwendungsprogrammierer stehen hierzu Klassen zur Verfügung, welche die Unterteilung in Module widerspiegeln und die Kombination von Modulen (*clustering*) unterstützen. Die Einfachheit der RACER-Schnittstelle bedingt, dass die Klassen, welche die Module repräsentieren, lediglich vier Methoden besitzen: *SendData(ULONG Size; UCHAR* Buffer)*, *CopyData(ULONG Size; UCHAR* Buffer)*, *MoveData(ULONG Size; UCHAR* Buffer)* und *ReceiveData(ULONG& Size; UCHAR* Buffer)*, wobei die ersten drei Methoden verschiedene Varianten zum Senden von Daten bereitstellen. Beim Senden der Daten wird ein Puffer übergeben, der die zu sendenden Daten enthält. Durch den Parameter *Size* wird die Größe des aus diesem Puffer zu übertragenden Datenblocks angegeben. CopyData legt eine Kopie der übergebenen Daten an, wobei SendData und MoveData den übergebenen Puffer verwenden. MoveData gibt jedoch im Gegensatz zu SendData den für den Puffer reservierten Speicher nach der Übertragung der Daten selbständig wieder frei.

Der Aufruf der Methoden ist nicht blockierend, das heisst, der Aufruf kehrt unmittelbar zurück und der Transfer der Daten findet im Hintergrund statt. Beim Aufruf von CopyData blockiert die Methode lediglich für die Zeit, in welcher der Puffer kopiert wird.

Beim Empfang der Daten muss ein ausreichend großer Puffer bereitgestellt und die gewünschte Datenmenge angegeben werden. Die Datenmenge ist dabei durch die Größe der vom Treiber verwendeten Ringpuffer beschränkt. In der vorliegenden Implementierung des Treibers entspricht die Ringpuffergröße der Größe der Modul-FIFOs. Die Empfangsmethode ist insofern nicht blockierend, als dass entweder die geforderte Datenmenge oder die zum Zeitpunkt des Aufrufs verfügbare Datenmenge

zurückgeliefert wird. Der Parameter *Size* enthält, nachdem die Methode zurückgekehrt ist, die tatsächlich in den Buffer übertragene Datenmenge.

Da weder die Sende- noch die Empfangsmethode blockierend ist, kann mit dieser Anwenderschnittstelle eine maximale Nebenläufigkeit realisiert werden.

Statusinformationen über die Konfiguration und den aktuellen Zustand der Karte werden dem Anwendungsprogrammierer durch eine entsprechende Klasse *RacerControl* angeboten. Diese Klasse unterstützt explizit nicht die Methoden *SendData()* und *ReceiveData()*, da die Kommunikation mit dem Controller lediglich über die Mailboxen des PCI-Kontollers abgewickelt wird und eine entsprechende Aufbereitung der Daten vonnöten ist. Es steht vielmehr ein beschränkter, aber erweiterbarer Satz von Befehlen (Tabelle 10) zur Verfügung, die vom Controller akzeptiert und verarbeitet werden.

Methodenname	**Funktion**
CODE_INIT_RING	Durch diesen Befehl wird die Hardware initialisiert. Im Zuge dieser Initialisierung werden auch die physikalischen Adressen der zusammenhängenden none-paged-Buffer durch Mailbox 2-4 übertragen, die durch den Treiber für jedes Modul angelegt werden müssen.
CODE_INIT_MID	Durch diesen Befehl wird die physikalische Adresse des zusammenhängenden none-paged-Buffer für die Statusinformationen der Hardware in Mailbox 2 übertragen.
CODE_MID_REQ	Durch diesen Befehl wird der Mastertransfer der Statusinformationen in den dafür vorgesehenen Buffer angestoßen.
CODE_INT_MASK	Durch diesen Befehl wird die Interrupt-Maske der Add-On Hardware übertragen. Es soll dadurch möglich werden, die Hardware an spezielle Anforderungen anpassen zu können.
CODE_FIFO_TRIGGER	Durch diesen Befehl werden die FIFO-Trigger gesetzt. Dies geschieht global, deshalb entfallen auch die Regions- und In/Out- Parameter.
CODE_SET_FIFO_OFFSET	Der Befehl setzt die FIFO-Offsets. Aufgrund der zusätzlichen Parameter lässt sich jede FIFO einzeln ansprechen.
CODE_CLUSTER	Durch diesen Befehl werden die Module geclustert. Der Region-Parameter wird folgendermaßen ausgewertet: 00 – kein Clustern 01 – Modul 1 und Modul 2 werden geclustert 10 – Modul 2 und Modul 3 werden geclustert 11 - alle 3 Module werde geclustert
CODE_READ_BUFFER	Hiermit wird der Hardware mitgeteilt, dass aus dem DMA-Buffer gelesen wurde. Als Parameter werden die Region und der aktuelle Adress-Pointer übergeben.
CODE_READ_REG	Der Befehl stellt eine Leseanforderung an die internen Register der Add-On Hardware dar. Der Parameter bezeichnet das zu lesende Register. Der Befehl ist primär für die interne Fehlersuche vorgesehen.
CODE_WRITE_BUFFER	Hiermit wird dem Host mitgeteilt, dass Daten in den DMA-Buffer geschrieben wurden. Die Anzahl der geschriebenen Bytes wird durch einen zusätzlichen Parameter übergeben.
CODE_FIFO_READY	Hierdurch wird dem Host mitgeteilt, dass die eingangsseitigen

	FIFOs bereit sind, neue Daten aufzunehmen. Je nach Konfiguration haben die FIFOs dann einen bestimmten Füllstand unterschritten oder sind leer. Als Parameter wird die entsprechende Region übergeben.
CODE_ERROR	Dieser Code dient zur Fehlerkontrolle und liefert einen Errorcode.
CODE_WRITE_REG	Dieser Code ist die Reaktion der Hardware auf eine CODE_READ_REG Anforderung. Sie liefert zusätzlich die Register-Identifikation und den Inhalt des Registers.
CODE_PT_TIMEOUT CODE_AMBEF_TIMEOUT CODE_AMBEF_GIVEUP_TIMEOUT	Diese Befehle stellen Time-outs für den Zugriff auf verschiedene Register der Hardware ein, damit im Betrieb eine Blockade vermieden werden kann.

Tabelle 10: Liste der RACER-Befehle

Der Anwendungsprogrammierer kann sich mit Hilfe des Befehlssatzes sowohl über den Füllstand der FIFOs eines jeden Moduls in dem durch die Hardware gesteckten Rahmen informieren als auch über die aktuelle Modulkonfiguration (*clustering*) und die Art der installierten Module. Darüber hinaus können Leistungsdaten ausgewertet werden, die von der Hardware bereitgestellt werden.

6.1.3.2. Die Softwareschichten des Gerätetreibers

Der Gerätetreiber besteht aus zwei Teilen. Ein Teil läuft im so genannten Kernelmodus ab: der *Kernel-mode*-Treiber. Den zweiten Teil bildet die Anwenderschnittstelle, welche in Form einer dynamisch gebundenen Bibliothek (DLL) vorliegt. Sie bietet einem Anwendungsprogrammierer den Zugriff auf die RACER-Hardware über C++ Objekte.

6.1.3.2.1. Kernel-mode-*Treiber*

Der *Kernel-mode*-Treiber beinhaltet Routinen, die beim Systemstart, oder wenn das System zurückgesetzt wird, den PCI-Bus nach einer RACER-Karte absuchen. Sofern die RACER-Hardware aufgefunden wird, initialisiert sie der *Kernel-mode*-Treiber. Diese Initialisierung beinhaltet das Mapping der gerätespezifischen Register, Interrupts und Speicherbereiche. Ferner werden durch den *Kernel-mode*-Treiber zusammenhängende, nicht virtuelle Speicherbereiche für die Ringpuffer der einzelnen Module angelegt. Der Hardware werden die Startadressen und Größen dieser Ringpuffer übermittelt. In der Hardware werden damit die Register für den DMA-Controller initialisiert. Die Größe der Ringpuffer ist so gewählt, dass sie den Inhalt eines ganzen FIFOs aufnehmen können (512kB). Abschließend wird ein Name in den Win32-Namensraum exportiert, unter welchem das Gerät RACER angesprochen werden kann. Ist die Initialisierung erfolgreich abgeschlossen, stellt der *Kernel-mode*-Treiber die grundlegenden Routinen zur Ansteuerung der RACER-Karte zur Verfügung. Auf dieser für Geräte üblichen Schnittstelle setzt die Anwenderbibliothek auf.

6.1.3.2.2. Anwenderbibliothek

Die Bibliothek, welche die eingangs beschriebene Anwenderschnittstelle implementiert, stellt den blockadefreien Datenstrom zur und von der RACER-Hardware sicher. Hierfür ist es insbesondere notwendig, große Datenpakete, die der Anwender auf die Karte übertragen möchte, in kleinere Pakete zu zerlegen, da unter allen Umständen die Be-

schickung eines bereits vollen Modul-FIFOs vermieden werden muss. Andernfalls würde der *Kernel-mode*-Treiber in dem geforderten Transfer blockieren, bis wieder ausreichend Platz im betreffenden FIFO verfügbar ist. Eine Ausführung paralleler Transfers in andere Module, bei denen noch eine entsprechende Kapazität vorhanden ist, wäre so nicht möglich.

Es werden daher immer nur Datenpakete transferiert, die maximal halb so groß wie ein Modul-FIFO sind. Die Hardware übermittelt dem Treiber jeweils einen Interrupt, wenn nach einem Transfer der Füllstand kleiner oder gleich der halben FIFO-Größe ist. Sie signalisiert damit die Bereitschaft, ein weiteres Datenpaket aufnehmen zu können. Vor jedem Transfer eines Datenpakets wird von der Anwenderbibliothek überprüft, ob der FIFO in der Lage ist, die Daten aufzunehmen, indem die entsprechende Information aus dem Treiber gelesen wird.

Dieser Mechanismus kann bei Bedarf anders konfiguriert werden, da für bestimmte Applikationen eine andere Strategie zu höheren Datendurchsätzen führen kann. So könnten zum Beispiel immer eine komplette FIFO-Größe oder nahezu beliebig große Datenpakete zur Karte transferiert werden, wenn das betreffende Modul mit jedem PCI-Takt ein Wort aus dem FIFO verarbeitet, also der FIFO im Grunde immer nur ein Wort für einen Takt zwischenspeichert. Die maximale Datenmenge hängt in einem solchen Fall von der Zahl der vom Modul generierten Ergebnisworte ab, da ein voller Ausgangs-FIFO zu einem Rückstau in den Eingangs-FIFO des Moduls führt. Generiert zum Beispiel das Modul für jedes eingehende Datenwort ein Ausgangswort, so kann die maximale Größe des Datenpakets der dreifachen FIFO-Größe entsprechen, da der Ausgangs-FIFO durch das weiter unten beschriebene Verfahren automatisch in den Ringpuffer des Treibers geleert wird, sofern dieser noch eine entsprechende Kapazität aufweist.

Damit mehrere Transfers sowohl für ein Modul als auch für verschiedene Module kurz nacheinander gestartet werden können, werden für jeden Methodenaufruf, der einen solchen Transfer auslöst (SendData, CopyData oder MoveData), die den Transfer beschreibenden Parameter[58] in eine Warteschlange eingereiht. Für jedes Modul existieren eine FIFO-Warteschlange und ein sie verwaltender Thread. Der verwaltende Thread entnimmt jeweils den ältesten Eintrag aus der Warteschlange und startet den eigentlichen Schreib-Thread, welcher den Transfer ausführt. Schreib-Threads für unterschiedliche Module laufen konkurrierend ab.

Der Rücktransfer der Daten läuft vollständig nebenläufig, da die RACER-Hardware nach zuvor einstellbaren Kriterien die Ausgangs-FIFOs der Module entleert. Es kann auf alle durch den FIFO bereitgestellten und programmierbaren FIFO-Flags reagiert werden. Erfüllen mehrere Ausgangs-FIFOs die Kriterien für einen Rücktransfer, so werden die Daten reihum aus den FIFOs entleert, wobei jeweils maximal eine komplette FIFO-Füllung transferiert wird. Ein Rücktransfer ist natürlich nur möglich, sofern der zugehörige Ringpuffer im Treiber noch eine hinreichende Kapazität aufweist. Ist ein Ringpuffer vollständig gefüllt und wird er nicht entleert, so führt dies zu einem Rückstau der Daten bis hin zum Eingangs-FIFO des betreffenden Moduls. Es obliegt dem Anwender, den Ringpuffer rechtzeitig zu entleeren, um einen solchen Rückstau zu vermeiden. In einer solchen Situation können durchaus weitere Datentransfers an das im Rückstau befindliche Modul initiiert werden. Jedoch werden vom Modul keine Daten mehr verarbeitet, solange der Ausgangs-FIFO nicht entleert wird.

---

[58] Die Parameter sind das Zielmodul, die Startadresse der zu transferierenden Daten und die Größe des zu transferierenden Datenpakets.

Der Treiber wird nach jedem DMA-Transfer in einen Ringpuffer über die transferierte Datenmenge durch einen Interrupt von der Hardware in Kenntnis gesetzt. Im Treiber werden daraufhin die lokal geführten Schreib- und Lesezeiger auf den Ringpuffer aktualisiert. Ebenso wird die Hardware mit Hilfe eines Interrupts darüber informiert, sobald Daten durch den Anwender aus dem Ringpuffer gelesen wurden. Außerdem findet ein Abgleich der lokalen Zeiger in der Hardware statt.

Neben den Datentransferfunktionen bietet die Anwenderbibliothek außerdem Statusinformationen über die Klasse RacerControl an. Diese geben Aufschluss über den aktuellen Zustand der Karte. Zudem kann das Verhalten des Kartencontrollers über diese Klasse konfiguriert werden. Die wesentlichen Statusinformationen umfassen dabei:

- den Füllstand der Ein- und Ausgangs-FIFOs (FIFO-Flags)
- die aktuellen Werte der Schreib- und Lesezeiger der Ringpuffer
- die Basisadressen der Ringpuffer und des Konfigurationsdatenbuffers
- die gewählten Auslöseschwellen (FIFO-Flags) für den Rücktransfer der Daten
- die Belegung der Modulsockel (Modul eingesteckt oder nicht)
- die Identifikationsdaten der Module (MID)
- die Einstellung der die Module verbindenden Switches (*clustering*)

Darüber hinaus stehen noch weitere Informationen zur Dauer, Größe und Anzahl der durchgeführten Transfers und zur Konfiguration von Abbruchkriterien (*time outs*), die vom Controller verwendet werden, zur Verfügung. Eine detaillierte Übersicht über diese Informationen und Konfigurationsmöglichkeiten findet sich im Anhang dieser Arbeit.

6.2. Leistungsbewertung der Hardware

Im Rahmen des SPIKELAB-Projekts entstanden die Arbeiten [Flieck2000], [Flieck2000], [Werkhausen2002] und [Bubolz2002] in denen VHDL-Implementierungen einer Ereignisliste und eines Neuronenmodells sowie deren Einbindung in den Simulatorkern von SPIKELAB entstanden.

6.2.1. Ereignisliste

In [Flieck1999] konnte mit Hilfe von Simulationen nachgewiesen werden, dass die in Hardware ablaufende Ereignislistenverwaltung ca. 10% bzw. 20%[59] schneller ist als eine Softwareimplementierung auf einem Intel Celeron Prozessor, der mit einem Takt von 417 MHz betrieben wurde. Da die Hardware mit einem Takt von 33 bzw. 40 MHz betrieben wird, ergibt sich zusätzlich ein Taktverhältnis von 12,6 bzw. 10,4 zwischen Software und Hardware. In Fliecks Simulationen wurden jedoch die Kosten für den Datentransfer vom Host zum RACER nicht berücksichtigt, so dass sich unter realen Bedingungen die Verhältnisse deutlich zu Gunsten der Softwaresimulation verschieben.

In [Werkhausen2002] wurden daher Messungen mit einer stark vereinfachten idealisierten Hardware-Implementierung durchgeführt. Die Implementierung emuliert das Verhalten eines in Hardware implementierten Calendarqueue-Algorithmus, welcher in der

[59] 10% bei einem Hardwaretakt von 33 MHz und 20% bei einem Hardwaretakt von 40 MHz.

Lage ist, mit jedem Modultakt ein Ergebnis zu generieren. Abschätzungen haben zudem gezeigt, dass eine solche Leistungsfähigkeit auch durch eine voll funktionale Implementierung erreicht werden kann. Die Messungen aus Abbildung 73 entsprechen den Messungen aus Abschnitt 3.1.1.2.4 (Abbildung 19).

Diese Messungen zeigen, dass bei Berücksichtigung des Datentransfers erst für sehr stark gefüllte Ereignislisten (ab 150.000 Ereignissen) die Hardware-Implementierung bei einer Growing-Sequenz leistungsfähiger ist als eine STL Prioritätswarteschlange. Bei der Deminishingsequenz werden die gleichen Laufzeiten erreicht, jedoch liegen hier die Laufzeiten beider Softwareimplementierungen deutlich unter der Laufzeit der Hardware-Implementierung. Die Softwareimplementierung der Calendarqueue ist in beiden Fällen schneller als die idealisierte Hardware-Implementierung. Aus diesem Grunde wurde von einer voll funktionalen Implementierung des Calendarqueue-Algorithmus abgesehen.

Eine Beschleunigung der Simulation durch die Ausführung der Ereignisliste in Hardware alleine ist in dieser Form nicht möglich, da die Gesamtleistung durch den Datentransfer bestimmt wird und die Zugriffe auf die Ereignisliste während der Simulation mit einer sehr hohen Frequenz stattfinden. Eine Bündelung dieser Zugriffe ist jedoch nicht möglich.

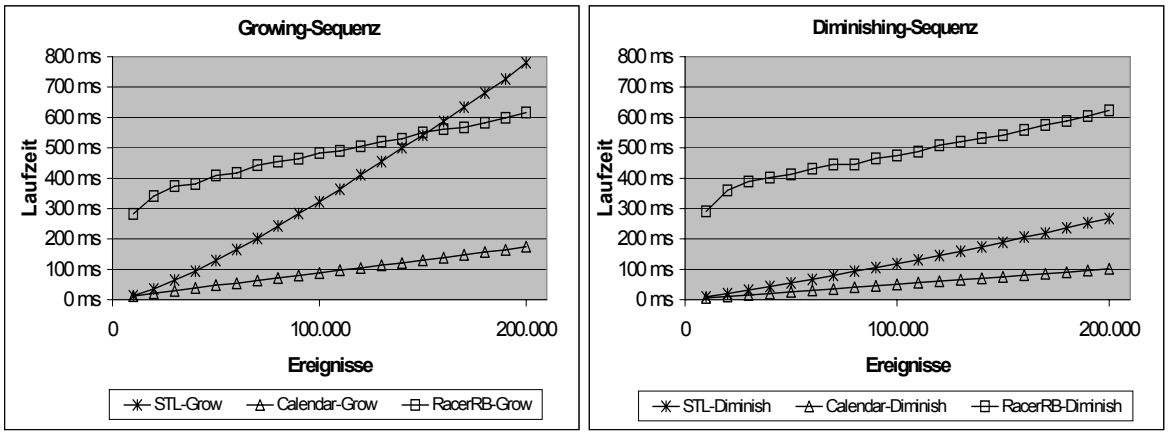


Abbildung 73: Messungen mit idealisierter Hardware-Implementierung des Calendarqueuealgorithmus aufgetragen über die Anzahl der in der Queue enthaltenen Ereignisse.

6.2.2. Neuronenmodelle

Simulationen in [Flieck2000] haben gezeigt, dass die Hardware-Implementierung eines Neuronenmodells 18% schneller ist als die Softwaresimulation auf einem Intel Pentium-II System mit 350 MHz Prozessortakt. Auch bei diesen Simulationen wurde der Datentransfer vom Host auf die RACER-Hardware nicht berücksichtigt.

Die theoretische Simulationsgeschwindigkeit der Hardware-Implementierung aus [Flieck2000] liegt im ungünstigsten Fall zwischen 7 und 204 Modultakten, das heißt also zwischen 0,231 µs und 6,732 µs[60]. Die kürzeste Laufzeit ergibt sich für das zweite Datenwort, wenn zwei aufeinander folgende Datenworte für das gleiche Neuron bestimmt sind und das Neuron eindeutig unterschwellig bleibt. Die längste Laufzeit ergibt sich, wenn von einem Datenwort auf das nächste ein Neuronenwechsel stattfindet und

[60] Die Zeitwerte gelten für einen Hardwaretakt von 33 MHz

der Schnittpunkt mit dem Schwellenwert bestimmt werden muss. Beschickt man das Hardwaremodell mit einzelnen Ereignissen, so ergibt sich überwiegend eine Laufzeit von 127 bis 150 Takten, also zwischen 4,191 µs und 4,950 µs pro verarbeiteten Puls. Ausgehend von 4,5 µs im Mittel ergibt sich somit eine theoretische Simulationsleistung von ca. 223.000 Ereignissen pro Sekunde – vorausgesetzt, dass sich der Datentransfer ideal mit der Berechnung auf dem Modul überlappt und nicht länger dauert als diese.

Abbildung 74 zeigt Messungen mit dem in der SPIKELAB-Simulation integrierten Hardware-Neuronenmodell aus [Flieck2000] im Vergleich zur reinen Softwaresimulation. Die Messungen wurden mit unterschiedlichen Simulationsmodi durchgeführt, die sich jeweils durch die Art der Ereignisverwaltung unterscheiden. Im einfachsten Fall wurden die Ereignisse einzeln verarbeitet (StdModIAFi), dann wurden alle Ereignisse innerhalb eines sicheren Horizonts gebündelt verarbeitet (StdModIAFi-Multievents bzw. Racer-Multievents) und schließlich erfolgte eine spezielle Bündelung der Ereignisse für die Verarbeitung in der RACER-Hardware (Racer-processRacer). Die Messungen fanden auf einem Intel Pentium III System mit einem Prozessortakt von 500 MHz und einem Frontbustakt von 100 MHz statt. Die speziell für die Hardware modifizierte gebündelte Verarbeitung der Ereignisse nähert sich für große Netze an die theoretisch bestimmte Leistungsfähigkeit der Hardware an und übertrifft die Leistungsfähigkeit der Softwaresimulation um 35%.

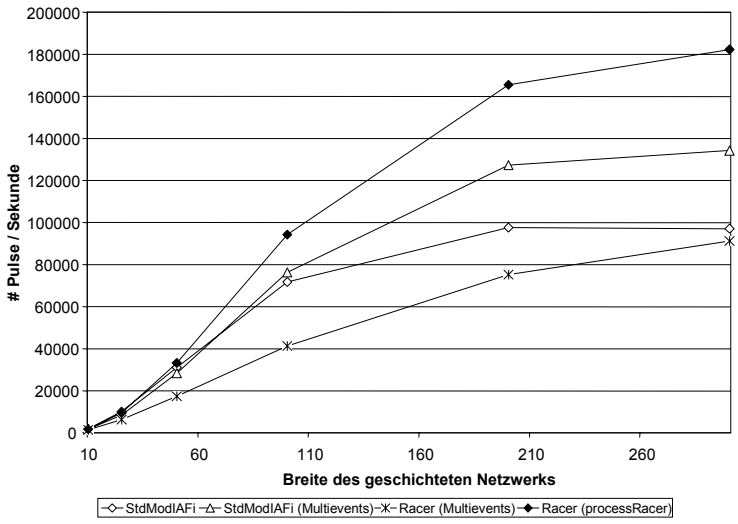


Abbildung 74: Simulationsleistung der Softwareimplementierung und der Hardware-Implementierung, mit und ohne gesammelte Verarbeitung der Ereignisse (aus [Bubolz2002]).

Die Berechnung der Neuronenmodelle lässt sich im Vergleich zur Berechnung in Software auf heutigen Rechnern beschleunigen. Dies ist jedoch nur der Fall für eine gebündelte Verarbeitung der Ereignisse.

Da die Verknüpfungen der Neuronen auf dem Host gespeichert werden, ist es möglich, unter Verwendung eines 64 MB Speichermoduls bis zu 1.040.384 Neuronen auf dem Modul zu halten und mit der angegebenen Leistung zu simulieren.

6.3. Ergebnisse und Folgerungen

Aus den Messungen wird deutlich, dass die Beschleunigung von Teilen der Simulation nur bedingt sinnvoll ist, da in allen betrachteten Beispielen die Leistung durch den Datentransfer vom Host auf die RACER-Hardware bestimmt wird, und dies, obwohl die Hardware, wie Abbildung 75 zu entnehmen ist, bereits in der vorliegenden Version einen Datentransfer mit einer hohen Bandbreite ermöglicht. Die mit DLL benannte Messkurve steht für die Bandbreite, die dem Benutzer effektiv zur Verfügung steht, da hier die Zeit gemessen wurde, die vom Aufruf der Schnittstellenfunktion bis zum Abschluss der Übertragung verstrichen ist. Die mit PT bezeichnete Messkurve gibt hingegen die in der Hardware gemessene Bandbreite an, da die Zeitdauer ab Beginn des Transfers in der Hardware gemessen wurde. Der Unterschied zwischen DLL und PT ergibt sich also alleine durch die Zeit, die von dem Aufruf in der Softwareimplementierung konsumiert wird. Die mit MT benannte Messkurve steht für die in Hardware gemessene Bandbreite für den Mastertransfer der Daten von der RACER-Hardware in den Speicher des Hosts.

Die Latenzen der einzelnen Transfers liegen bei 125,14 µs beim DLL-Aufruf, 15,99 µs PT-Hardware-Transfer und 2,69 µs Mastertransfer.

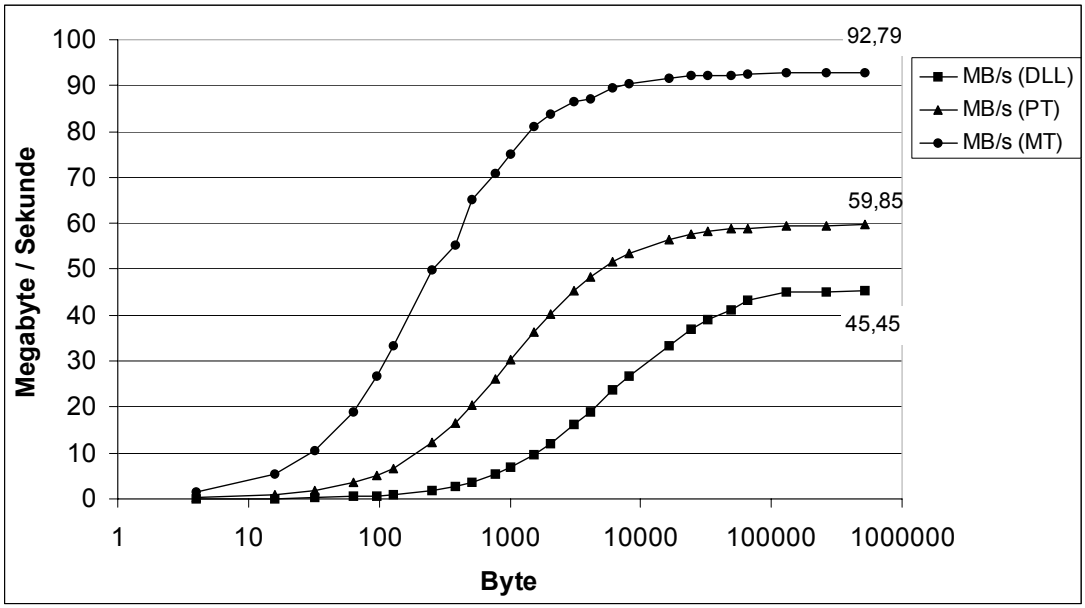


Abbildung 75: Bandbreite für Datentransfers vom Host zur RACER-Hardware (DLL und PT) und von der RACER-Hardware in Speicher des Hosts (MT). (DLL, PT und MT – siehe Text)

Es liegt die Vermutung nahe, dass durch die Bereitstellung des Mastertransfers für den Datentransfer zur RACER-Hardware eine noch höhere Bandbreite und eine geringere Latenz erreicht werden kann. Hierbei ist jedoch zu berücksichtigen, dass entgegen dem Mastertransfer von der RACER-Hardware zum Host für die umgekehrte Richtung der Transfer immer durch den Host, mittels eines Mailboxinterrupts, initiiert werden muss. Daher ist nicht davon auszugehen, dass sich die Latenz reduziert, sondern dass sie sich im ungünstigsten Fall sogar noch vergrößert. Dennoch wäre durch diese Art des Transfers das Hostsystem bei größeren Datenpaketen stärker entlastet, so dass die Nebenläufigkeit durch die RACER-Hardware besser ausgenutzt werden könnte.

Aufgrund der angestellten Untersuchungen würde es sich anbieten, sowohl die Ereignisliste als auch die Neuronenmodelle gemeinsam auf der Hardware zu berechnen, indem auf dem RACER-Modul ein kompletter Subsimulator implementiert wird, in welchem

die Optimierungen der Hardware-Implementierungen aus [Flieck1999] und [Flieck2000] einfließen. Mit dem zurzeit verfügbaren Modul, welches mit einem FPGA mit 200.000 Gatteräquivalenten bestückt ist, ist eine solche Implementierung jedoch nicht möglich. In Kürze erschwingliche FPGA-Bausteine mit 1000.000 Gatteräquivalenten böten diese Möglichkeiten hingegen sehr wohl. Mit derartigen Modulen ließe sich eine deutliche Beschleunigung gegenüber der reinen Softwaresimulation auf aktuellen Systemen erreichen, da sich eine vollständig in Hardware ablaufende Verteilung der Simulation realisieren ließe. Eine solche Simulation würde eine wesentlich effizientere Kommunikation aufweisen als eine über ein Netzwerk verteilte Simulation. Zudem kämen in dieser Simulation nachgewiesenermaßen schnellere, hardwarebeschleunigte Subsimulatoren zum Einsatz.

Eine weitere, hier nicht weiter untersuchte Einsatzmöglichkeit der RACER-Hardware ist die Einbindung digitaler und analoger Hardware, welche Pulse für die Simulation bereitstellt. Insbesondere in den Fällen, in denen kontinuierliche Signale verarbeitet werden müssen, bietet es sich an, die Vorverarbeitung bzw. Umwandlung in Pulse auf einem RACER-Modul vorzunehmen und anschließend die Pulse in die Simulation einzuspeisen (vgl. Abschnitt 3.2.3).

Kapitel 7 - Zusammenfassung und Ausblick

Eigenschaften pulsverarbeitender neuronaler Netze, wie eine spärliche Vernetzung und eine geringe Aktivität, legen eine ereignisgetriebene Simulation dieser Netzwerke nahe. Dennoch werden in der Regel Verfahren verwendet, die auf einer Zeitscheibensimulation basieren, da eine Implementierung auf digitalen Rechenwerken meist leichter fällt. Eine konsequent ereignisgetriebene Simulation erfordert es, umzudenken, da die Simulation nicht mehr in äquidistanten Schritten voranschreitet, jedoch für die Simulation pulsverarbeitender neuronaler Netze eine höhere Simulationsleistung bietet, denn die Simulationszeit hängt lediglich von der geringen Aktivität in diesen Netzwerken ab.

Durch diese Arbeit entstand das Simulationssystem SPIKELAB (→ Kapitel 4 - Das SPIKELAB Simulationssystem), in dem eine ereignisgetriebene und verteilte Simulation implementiert wurde.

Im Vergleich zu GENESIS, dem am häufigsten eingesetzten Zeitscheibensimulator für pulsverarbeitende neuronale Netzwerke, konnten mit der ereignisgetriebenen Simulation von SPIKELAB 7 bis 600-fach kürzere Simulationszeiten erreicht werden (→ Abschnitt 4.3.1). Durch die Untersuchungen wurde auch deutlich, dass eine feinere Zeitauflösung nahezu keinen Einfluss auf die Simulationsleistung der ereignisgetriebenen Simulation hat, während feinere Zeitauflösungen bei der Zeitscheibensimulation mit untragbar hohen Laufzeiten erkauft werden müssen. Insofern ist das vorgestellte Simulationsverfahren besser als ein Zeitscheibenverfahren für eine präzise Untersuchung der zeitabhängigen Informationsverarbeitung in pulsverarbeitenden neuronalen Netzen geeignet.

Die Simulation großer Netzwerke wird durch eine Verteilung beschleunigt, oder überhaupt erst möglich. SPIKELAB nutzt auch bei der verteilten Simulation die Eigenschaften pulsverarbeitender neuronaler Netze. Eine dezentrale Synchronisation trägt hier sowohl der geringen Aktivität als auch der spärlichen Vernetzung solcher Netzwerke Rechnung. Simulationen mit beispielhaften Testnetzwerken (→ Abschnitt 4.3.3.2) zeigen, dass der im Vergleich zur zentralen Synchronisation höhere Aufwand für eine dezentrale Synchronisation nicht ins Gewicht fällt, so dass die Verteilung der Simulation bereits für ein einfaches, konservatives Synchronisationsverfahren, welches in der vorliegenden Implementierung verwendet wurde, eine effiziente Verteilung der Simulation erlaubt. Zudem skalieren Simulationen mit einem solchen Synchronisationsverfahren grundsätzlich besser, da nicht wie bei einer zentralen Synchronisation von jedem Rechner eine Kommunikation mit einer zentralen Verwaltung notwendig ist. Den Messungen aus Abschnitt 4.3.3.2 ist zu entnehmen, dass die Simulation selbst noch im ungünstigsten Fall, nämlich der Verteilung eines vollvernetzten Netzwerks auf mehreren Rechnern, gut skaliert.

Mit Hilfe umfangreicher Messungen (→ Abschnitt 3.4) wurde eine quantitative Grundlage für die Auswahl eines geeigneten Nachrichtensystems zur Verteilung der ereignisgetriebenen Simulation gelegt. Basierend auf diesen Untersuchungen bot sich CORBA

an, um unterschiedliche Rechner in die Simulation einzubinden und somit verfügbare Rechenleistungen zu nutzen.

Aufgrund der vorgestellten Ergebnisse bietet es sich an, die prinzipiellen Vorteile des ereignisgetriebenen Simulationsverfahrens durch eine behutsame Optimierung im Hinblick auf eine speichereffizientere Implementierung weiter zu verbessern, ohne dabei die zur Zeit verfügbare Flexibilität der Simulation einzuschränken. Hierdurch würde die Simulation noch größerer Netze möglich.

Bei der verteilten Simulation bietet es sich an, zu untersuchen, ob die ohnehin leistungsfähige Simulation durch den Einsatz optimistischer Synchronisationsverfahren noch weiter verbessert werden kann. Darüber hinaus würde sich eine Portierung des Simulators auf das Message-Passing-Interface (MPI) anbieten, um Untersuchungen mit massiv parallelen Rechnern zu ermöglichen.

Sollen Simulationen unter Echtzeitbedingungen ablaufen, so reicht eine reine Softwaresimulation in der Regel nicht mehr aus und es ist notwendig, Teile der Simulation oder die gesamte Simulation in Hardware zu beschleunigen. Da bei einer Hardware-Implementierung jedoch immer Einschränkungen in der Flexibilität der Simulation hingenommen werden müssen, sei es bei der Berechnungsgenauigkeit, beim Grad der Vernetzung oder der Variabilität einzelner Parameter, sollte eine Hardware-Implementierung flexibel genug sein, um Variationen der in Hardware realisierten Komponenten zuzulassen.

Mit der RACER-Hardware wurde im Rahmen dieser Arbeit ein solch flexibles Hardwaresystem entwickelt und daran untersucht, wie sich Teile der Simulation damit beschleunigen lassen. Besonderes Augenmerk lag dabei einerseits auf einem einfachen Interface zur Hardware und andererseits auf einer schnellen und einfachen Konfigurierbarkeit des Systems. Entstanden ist ein System mit einem datenflussorientierten Interface zu austauschbaren Modulen, welche bis zu zwei programmierbare Bausteine tragen können. Der Konfigurationsmechanismus für die programmierbaren Bausteine auf den Modulen ist ein integraler Bestandteil der Architektur und erlaubt es, die Bausteine während der Laufzeit zu konfigurieren. Da die Module austauschbar sind, kann einerseits mit der Entwicklung programmierbarer Logik Schritt gehalten werden und andererseits können auch verschiedene andere programmierbare und nicht programmierbare Hardwarearchitekturen in das System eingebunden werden.

Untersuchungen (→ Abschnitt 6.2) zur Beschleunigung einzelner Teile der ereignisgetriebenen Simulation zeigen, dass eine Beschleunigung mit der aktuell vorliegenden Hardware wegen der Kapazitätsbeschränkung des verfügbaren FPGA, nicht immer gelingt. Es wird jedoch auch deutlich, dass mit komplexeren programmierbaren Bausteinen, auf denen sich ganze Subsimulatoren implementieren lassen, eine Verteilung der Simulation wie im Workstationcluster, aber mit einer deutlich leistungsfähigeren Kommunikation und optimierten Berechnungsknoten realisieren ließe. Die RACER-Karte kann dabei nach wie vor als Basis dienen. Darüber hinaus haben die Untersuchungen gezeigt, dass durch die prototypische Entwicklung mit der RACER-Hardware die - Beschleunigungspotentiale von Teilbereichen der Simulation einfach untersucht werden können und letztlich als Ausgangsbasis für eine Gesamtlösung in komplexeren programmierbaren Bausteinen oder auf hochintegrierten Schaltkreisen dienen können.

Kapitel 8 - Anhang

8.1. Installation des Softwarepakets SPIKELAB

8.1.1. Softwarepaket

Entpacken Sie das Softwarepaket in ein Verzeichnis (Installationsverzeichnis – z. B. C:\Spikelab). Unterhalb des gewählten Verzeichnisses befinden sich die Verzeichnisse *config*, *bin*, *lib*, *source* und *VC60*. Das Verzeichnis *source* hat die Unterverzeichnisse *Common*, *OMNIThread*, *ParameterManager*, *Partitioner*, *RACER-Driver*, *Randlib*, *Simulator*, *Spikelab-GUI*, *Spikelab-Utility* und *Utility-Progs*.

8.1.2. Umgebungsvariablen

Setzten Sie die Umgebungsvariable SPIKELAB_ROOT = [Installationsverzeichnis]

Außerdem sollte auf dem System die Umgebungsvariable TEMP gesetzt sein.

8.1.3. Übersetzung

In dem Unterverzeichnis *VC60* befindet sich ein Workspace für das Microsoft Developer Studio 6.0. Laden Sie den Workspace in das Developer Studio, aktivieren Sie das Projekt Spikelab und stoßen Sie mit F7 der Buildprozess an. Nach Abschluss des Buildprozesses befindet sich die ausführbare Datei Spikelab.exe in dem Verzeichnis [Installationsverzeichnis]\bin\Debug oder [Installationsverzeichnis]\bin\Release, je nachdem, welche Konfiguration (Debug oder Release) gewählt wurde.

8.1.4. RSH Deamon

Für den verteilten Betrieb des Simulators wird auf PCs mit dem Betriebssystem Windows NT4 oder Windows 2000 noch ein zusätzlicher rsh-Deamon benötigt. An dieser Stelle sei in Vertretung für viele alternativ einsetzbare rsh-Deamons die Einrichtung des rsh-Deamons *atrls* der Firma *Ataman* (http://www.ataman.com) beschrieben, welcher als Evaluierungsversion bezogen werden kann.

Nachdem die Dateien des rsh-Deamons in ein Verzeichnis entpackt wurden, sollte dieses Verzeichnis in die Umgebungsvariable PATH mit aufgenommen werden. Anschließend muss der Deamon noch installiert und gestartet werden. Hierfür öffnet man eine DOS-shell und gibt als Administrator die Befehlsfolge „atrls install start“ ein. Ob die Installation erfolgreich war und der Dienst gestartet wurde, kann den Ausgaben entnommen werden.

Nach der Installation findet sich in der Systemsteuerung ein Icon in Form eines großen „A“ für den rsh-Dienst. Durch einen Doppelklick auf dieses Icon wird der Konfigura-

tionsdialog des Dienstes geöffnet. Hier müssen die Benutzer eingetragen werden und die Rechner, von denen sie mittels rsh auf den betreffenden Rechner zugreifen möchten.

8.1.5. CORBA

Die CORBA-Version des Simulators ist auf Basis des Object Request Brokers (ORB) von AT&T, OmniORB entwickelt worden. Dieser ORB kann für nicht kommerzielle Projekte kostenfrei bezogen werden:
http://www.uk.research.att.com/omniORB/omniORB.html.

Entpacken Sie das Softwarepaket in ein Verzeichnis (z. B. C:\Programme\omni). Gegebenenfalls ist außerdem eine aktuelle Version von PYTHON zu installieren. Tragen Sie das Binärverzeichnis in die Umgebungsvariable PATH ein. Anschließend ist der Nameservice zu installieren. Diese Prozedur ist nur auf einem der Rechner aus dem Netzwerk durchzuführen. Auf diesem sollte dann immer der Nameservice betrieben werden:

1. Legen Sie in dem Verzeichnis [Verzeichnis, in das OmniORB entpackt wurde]\config ein Unterverzeichnis mit dem Namen „Log“ an.
2. Öffnen Sie eine DOS-shell und geben Sie omninames –start 12345 ein. Hiermit wird der Nameservice initialisiert und generiert eine Ausgabe, welche die Interoperable Object Reference (IOR) des Nameservice enthält. Die Ausgabe könnte zum Beispiel folgendermaßen aussehen:

```
C:\Programme\omni\bin\x86_win32>omninames

Mon Aug 05 21:52:37 2002:

Read log file successfully.
Root context is
IOR:010000002b00000049444c3a6f6d672e6f72672f436f734e616d696e672f4e6
16d696e67436f6e746578744578743a312e3000000100000000000002700000001
0100000c0000003139322e3136382e312e310039304e610b0000004e616d6553657
276696365
Checkpointing Phase 1: Prepare.
Checkpointing Phase 2: Commit.
Checkpointing completed.
```

3. Kopieren Sie die IOR aus der Ausgabe heraus und fügen Sie diese in einen Editor ein. Es muss nun alles, beginnend mit „IOR:“ bis zum Ende der langen Zahlenkolonne kopiert werden. Entfernen Sie im Editor alle Zeilenumbrüche, so dass eine lange, zusammenhängende Zahlenkolonne entsteht, an deren Anfang "IOR:" steht.
4. Öffnen Sie den Registry-Editor (regedit.exe) und legen Sie unter dem Pfad HKEY_LOCAL_MACHINE\SOFTWARE einen Schlüssel „ORL“ an. Unterhalb dieses Schlüssels einen weiteren Schlüssel „omniORB“ und darunter den Schlüssel „2.0“. In dem nun vorhandenen Schlüssel HKEY_LOCAL_MACHINE\SOFTWARE\ORL\omniORB\2.0 fügen Sie eine Zeichenfolge ein und benennen diese „NAMESERVICE“; als Wert dieser Zeichenfolge fügen Sie die IOR aus ihrem Editor ein – inklusive des führenden „IOR:“.
5. Für die Konfigurierung anderer Rechner im Netzwerk, die in die Simulation eingebunden werden sollen, bietet es sich an, den kompletten Schlüssel HKEY_LOCAL_MACHINE\SOFTWARE\ORL in eine *.reg Datei zu exportieren. Tragen Sie anschließend diese Datei auf den anderen Rechnern per Doppelklick in die Registry ein. Der Rechner kann sich dann mit Hilfe der dort eingetragenen IOR

mit dem Nameservice verbinden, sofern dieser auf dem Rechner, auf dem die Initialisierung des Nameservice stattgefunden hat, gestartet wurde.

Um die CORBA-Implementierung in SPIKELAB zu übersetzen, müssen folgende Projekte aus dem Workspace übersetzt werden: CDistrSim, CDistrSim_UI_Omni, CStarterOmni und CSubSim_omni. Gegebenenfalls müssen zudem die DLLs aus den Verzeichnissen %SPIKELAB_ROOT%\bin\[Release|Debug] in die entsprechenden Verzeichnisse [Debug|Release] der entsprechenden CORBA Projekte kopiert werden.

8.2. Installation des RACER Hardware-Treibers unter NT4

Das Treiberpaket besteht aus folgenden Dateien:

- Racer.sys
- s5933lib.dll
- s5933lib.h
- s5933lib.lib

Die Installation des Treibers erfolgt in 5 Schritten:

1. Kopieren von Racer.sys in das Verzeichnis %SystemRoot%\SYSTEM32\DRIVERS
2. Kopieren von S5933lib.dll in das Verzeichnis %SystemRoot%\SYSTEM32
3. Eintragen von Startwerten unter dem Driver Service Key in der Registry
4. Booten des Systems, um den Service Control Manager vom neuen Treiber zu unterrichten
5. Wird der Treiber manuell gestartet, so wird hierfür das Applet „Dienste“ in der Systemsteuerung verwendet.

Erläuterungen zu Punkt 3:

Während des Bootvorgangs lädt Windows NT die Registry und extrahiert daraus verschiedene Werte. Unter anderem wird auch eine Liste von verfügbaren Treibern aufgebaut. Es ist daher wichtig, den Treiber durch einen entsprechenden Registry-Eintrag beim System anzumelden. Der Servicekey muss unter dem Pfad HKEY_LOCAL_MACHINE/System/CurrentControlSet/Services angelegt werden und hat den gleichen Namen wie der Treiber (ohne Endung). Unterhalb dieses Servicekeys sind Werte für den Typ, die Startart und das Verhalten für den Fall anzugeben, in dem das Laden des Treibers fehlschlägt. Folgende Tabelle fasst die betreffenden Werte zusammen:

Name	Datentyp	Wert	Beschreibung
Racer	(Key)		Der Name des Treibers ohne Endung (Racer.sys)
Type	REG_DWORD	0x1	Kernel-Mode-Treiber
		0x2	File-System-Treiber
Start	REG_DWORD	0x0	durch System Loader starten
		0x1	während der Initialisierung von NT starten
		0x2	nachdem das ganze System gestartet ist, starten
		0x3	manuell starten
		0x4	deaktiviert
ErrorControl	REG_DWORD	0x0	Error loggen aber ignorieren
		0x1	Error loggen und benachrichtigen
		0x2	Error loggen und rebooten

Der Treiber sollte möglichst spät oder manuell geladen werden. ErrorControl kann ganz nach Wunsch eingestellt werden.

Ist der Treiber gestartet, könnte schon direkt über eine Win32 Applikation auf ihn zugegriffen werden. Die Zugriffe sollten jedoch ausschließlich über die Funktionen der DLL s5933lib.dll erfolgen. Um eine Win32 Applikation zu übersetzen, die Gebrauch von dieser DLL macht, muss einerseits der Header s5933lib.h in den Quellcode eingebunden werden, andererseits muss die Datei 5933lib.lib durch den Linker zur Applikation gebunden werden.

Für einen direkten Zugriff auf den Treiber (unter Umgehung der DLL) muss der Header iocontrol.h in den Quellcode eingebunden werden. Ein solches Vorgehen empfiehlt ich jedoch nur, wenn die Funktionsweise von Treiber und DLL vollständig verstanden wurde.

8.3. Handhabung der RACER-Hardware

In Abbildung 76 sind die Konfigurationsschalter (DIP-Switches) der RACER-Hardware abgebildet. Die Schalter U57, U58 und U65 dienen einerseits dazu, den Typ des für die Programmierung des Kontroll-FPGAs und des PTU-FPGAs verwendeten EEPROMs einzustellen. Andererseits kann mit ihnen zwischen externer Programmierung und Programmierung über den EEPROM Baustein umgeschaltet werden. Die Programmierstecker der verschiedenen Bausteine befinden sich am linken Rand der RACER-Karte und sind mit der jeweiligen Bausteinnummer versehen.

Die Konfigurationsschalter U60 und U61 dienen dazu, den Default Offset für die „Almost“-Signale der FIFOs zu bestimmen. Es sind die Werte 127, 4095, 8191 und 16363 Worte gemessen vom vollen, respektive vom leeren Zustand des FIFOs. Die Offsets vom vollen und vom leeren Zustand ist immer identisch.

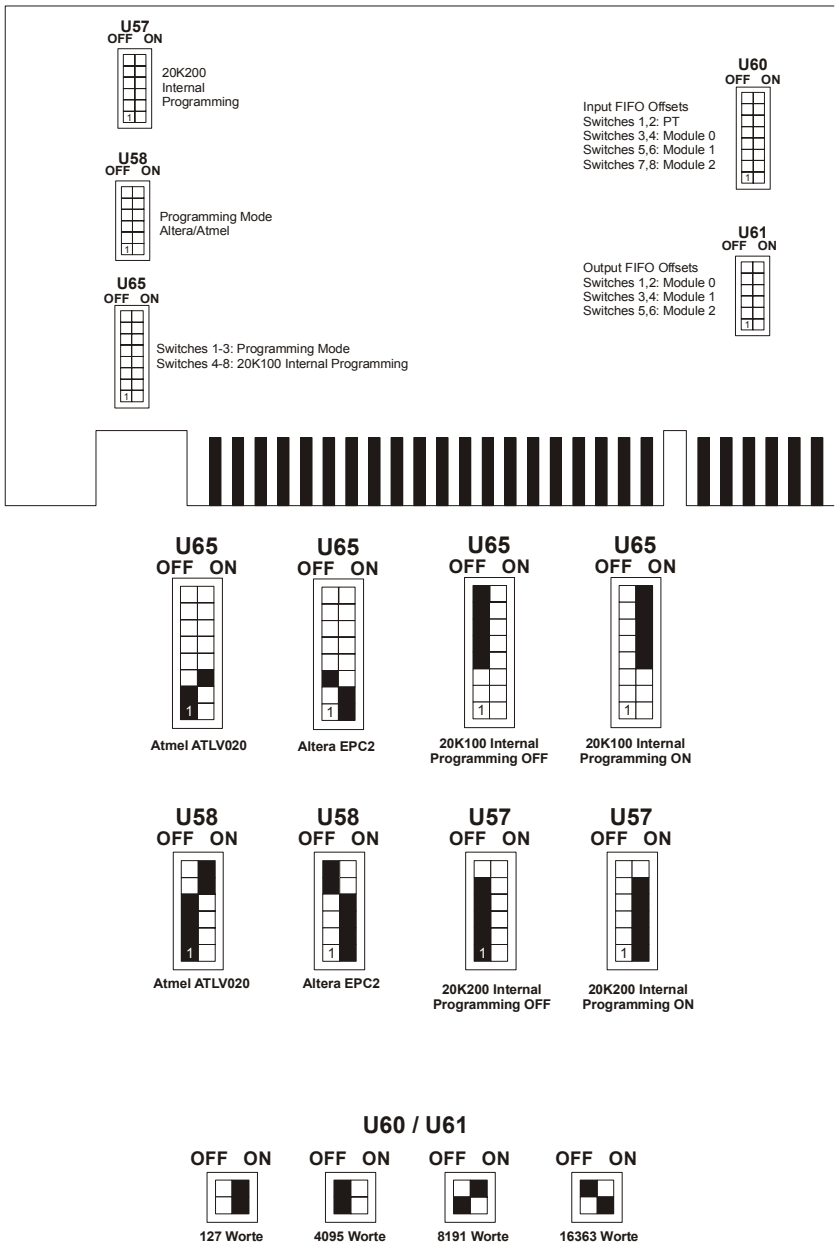


Abbildung 76: Lage und Belegung der Konfigurationsschalter (DIP-Switches) auf der RACER-Hardware

Kapitel 9 - Referenzen

[Abeles1982**]** Abeles Moshe, (1982). *Local Cortical Circuits: An Electrophysiological Study*, Springer-Verlag.

[Abeles1991**]** Abeles Moshe, (1991). *Corticonics: Neural Circuits of the Cerebral Cortex*, Cambridge University Press.

[Adrian1926**]** Adrian E.D., (1926). *The impulses produced by sensory nerve endings*. Part I, Journal of Physiology (London), Volume 61, S. 49-72.

[Adrian1932**]** Adrian E.D., (1932). *The Mechanism of Nervous Action: Electrical Studies of the Neurone*. University Pennsylvania Press, Philadelphia.

[Aertsen1993**]** Aertsen A. (ed.), (1993). *Brain Theory: Spatio-Temporal Aspects of Brain Function*. Elsevier.

[Alexander1999**]** Alexander J., (1999) . *Ein Navigations- und Analysetool für pulsverarbeitende neuronale Netze*. Diplomarbeit, Universität Bonn.

[BeeJanEck1990**]** Beerhold, J. R., Jansen, M.,and Eckmiller, R., (1990) *Pulse-processing neural net hardware with selectable topology and adaptive weights and delays*. In: Proc. IEEE Int. Joint Conf. on Neural Networks, San Diego, vol. 2, S. 569 – 574.

[BowBee1998**]** Bower J.M., Beeman D., (1994). *The Book of GENESIS*. Springer Verlag.

[Braitenberg1977**]** Braitenberg Valentin, (1977). *On the Texture of Brains*. Springer Verlag.

[BraSch1991**]** Braitenberg Valentin, Schüz A., (1991). *Anatomy of the Cortex: Statistics and Geometry*. Springer Verlag.

[BriClaDir1997**]** Brissinck W., Clarysse S., Dirkx E., (1997). *A Hierarchical Approach to Distributed Discrete Event Simulation*, Proc. of IASTED – International Conference on Parallel and Distributed Computing and Networks.

[Brown1988**]** Brown Randy, (1988). *Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem*. Communications of the ACM, Volume 31, S. 1220-1227.

[BroJoh1983**]** Brown Thomas H., Johnston Daniel, (1983). *Voltage-Clamp Analysis of Mossy Fiber Synaptic Input to Hippocampal Neurons*. Journal of Neurophysiology, Volume 50, S. 487-507.

[Bubolz2002**]** Bubolz Dietmar, (2002). *Integration hardwarebeschleunigter pulsverarbeitender Neuronenmodelle in eine Softwaresimulation*. Diplomarbeit, Universität Bonn, Bonn.

[CanEck1990**]** Canditt, S., and Eckmiller, R., (1990) *pulse coding hardware neurons that learn Boolean functions*. In: Proc. IEEE Int. Joint Conf. Neural Networks, Washington, vol. 2, S. 102 – 105.

[ChaMis1982**]** Chandy K. M., J. Misra, (1982) *Distributed simulation: A case study in design and verification of distributed programs*, IEEE Transactions on Software Engineering, SE-5(5):440-452.

[ConGutPri1982**]** Connors B.W., Gutnick M.J., Prince D.A., (1982). *Electrophysiological Properties of Neocortical Neurons in Vitro*. Journal of Neurophysiology, Volume 48, S. 1302-1321.

[Delorme1998**]** Delorme Arnaud, (1998) persönliche Kommunikation, Bonn.

[DomHemSch1995**]** Domany Eytan, Van Hemmen Leo, Schulten Klaus, (1995). *Models of Neural Networks*. Springer Verlag.

[EccIto1967**]** Eccles John C., Ito Masao, (1967). *The Cerebellum as a Neuronal Machine*. Springer Verlag.

[EckBauJor1988**]** Eckhorn R., Bauer R., Jordan W., Brosch M., Kruse W., Munk M., Reitboeck H. J., (1988). *Coherent Oscillations: A Mechanism of Feature Linking in the Visual Cortex*, Biological Cybernetics, Volume 60, S. 121–130, Springer Verlag.

[EckReiArn1989**]** Eckhorn R., Reitboeck H. J., Arndt M., Dicke P., (1989). *Feature Linking via Stimulus — Evoked Oscillations: Experimental results from Cat Visual Cortex and Functional Implications from a Network Model*. Proceedings of ICNN I, S. 723–730.

[Eckmiller1991**]** Eckmiller, R., (1991). *Pulse processing neural systems for motor control*. In: Artificial Neural Networks, Kohonen, T., Mäkisara, K., Simula, O., Kangas, J. (eds.), Elsevier, Amsterdam, vol. 1, S. 345 – 350.

[Eckmiller1993**]** Eckmiller, R., (1993). *Concerning the challenge of neurotechnology*. In: Neurobionics, Bothe, M.-W., Samii, M., and Eckmiller, R. (eds.), Elsevier, Amsterdam, S. 21 – 28.

[EckNap1993**]** Eckmiller, R., Napp-Zinn, H. (1993). *Information processing in biology-inspired pulse coded neural networks*. In: Proc. Int. Joint Conf. on Neural Networks, Nagoya, vol 1, S.643-648.

[Eckmiller1994**]** Eckmiller, R. (ed.) (1994). *Final Report of the Feasibility Study for a Neurotechnology Program*, BMFT, Bonn.

[Eckmiller1994a**]** Eckmiller, R., (1994). *Biology-inspired pulse processing neural networks (BPN) for neurotechnology.*In: Proc. ICANN '94, Sorrento, Springer Verlag, S. 1329 – 1334.

[Eckmiller1994b**]** Eckmiller, R., (1994). *Biology-inspired pulse processing neural nets with adaptive weights and delays - Concept sources from neuroscience versus applications in industry and medicine.* In: Computational Intelligence, Zurada, Marks II, Robinson (eds.), IEEE Press, New York, S. 276-284.

[Elman1990**]** Elman J. L. (1990). *Finding structure in time*. Cognitive Science, vol. 14, S. 179–211.

[EriLadLaM1994**]** Erickson K.B., Ladner R.E., LaMarca A., (1994). *Optimizing Static Calendar Queues*. Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science.

[FidMat1982**]** Fiduccia C.M., Mattheyses R.M., (1982). *A linear time heuristic for improving network partitions*. Proceedings 19. IEEE Design Automation Conference, S.175-181.

[FisPat1994**]** Fischer Michael J., Paterson Michael S., (1994). *Fishspear: A Priority Queue Algorithm*. Journal of the Association for Computing Machinery, 41.1, S 3-30.

[Flieck1999**]** Flieck Joachim, (1999). *Hardwarebeschleunigung einer Ereignislistenverwaltung*. Diplomarbeit, Universität Bonn, Bonn.

[Flieck2000**]** Flieck Michael, (2000). *Entwicklung eines Datenflußmoduls für einen PCI-Adapter zur Hardwarebeschleunigung eines Neuronenmodells*. Diplomarbeit, Universität Bonn, Bonn.

[FosShe1897**]** Foster M., Sherrington C., (1897). *A Textbook of Physiology, The Central Nervous System*, part three.

[FowSco1998**]** Fowler M., Scott K., (1998). *UML konzentriert*. Addison-Wesley, Bonn.

[Frank1997**]** Frank G., (1997). *Ein digitales Hardwaresystem zur echtzeitfähigen Simulation biologienaher neuronaler Netze.* Dissertation, Universität-GH Paderborn, HNI-Ver-lagsreihe, Bd. 26, Prof. Dr. rer. nat. Hartmann (Hrsg.), Paderborn.

[FraBilHar1995**]** Frank G., Bilau N., Hartmann G., (1995). *Hardware Accelerator zur Simulation puls-codierter Neuronaler Netze.* DAGM 1995, Bielefeld. In: Sagerer, G; Posch, S.; Kummert, F. (Hg.): Mustererkennung 1995. Informatik aktuell. Berlin u.a., Springer Verlag, S. 194-201.

[FraHar1995**]** Frank G., Hartmann G., (1995). *An Artificial Neural Network Accelerator for Pulse-Coded Model Neurons.* ICNN95, Perth, Australia. In: Proceedings on International Conference on Neural Networks, Perth, Volume 4, S. 2014-2018.

[FraHarJah1999**]** Frank G., Hartmann G., Jahnke A., Schäfer M., (1999). *An Accelerator for Neural Networks with Pulse-Coded Model Neurons.* IEEE Transactions on Neural Net-works, Special Issue on Pulse Coupled Neural Networks, Volume 10, S. 527-539.

[Fujimoto2000**]** Fujimoto Richard M., (2000). *Parallel and Distributed Simulation Systems*. Wiley & Sons, New York.

[GeiBegJia1994**]** Geist A., Beguelin A., Jiang W., Manchek R., Sunderam V., (1994). *PVM: Parallel Virtual Machine*. MIT Press.

[Gerstner1990**]** Gerstner W., (1990). *Associative memory in a network of 'biological' neurons*. Advances in Neural Information Processing Systems 3: S. 84-90.

[GerHem1992**]** Gerstner W., van Hemmen J.L., (1992). *Associative memory in a network of 'spiking' neurons*. Network 3: S. 139-164.

[GerHem1994**]** Gerstner W., van Hemmen J.L., (1994). *Coding and information processing in neural networks*. In: Models of neural networks II,

Domany E., van Hemmen J.L., Schulten K. editors, Springer Verlag, New York, S. 1-93.

[GerKemHem1996**]** Gerstner W., Kempter R., van Hemmen J., Wagner H., (1996). *A neuronal learning rule for sub-millisecond temporal coding*. Nature 383, 76 – 78.

[GlaSch1997**]** Glass G., Schuchert B., (1997). *Einführung in STL*. Prentice Hall, München.

[GraAnl1998**]** Grassmann Cyprian, Anlauf Joachim K. (1998). *Distributed, Event Driven Simulation of Spiking Neural Networks*. Neural Computation 98: 100-105.

[GraAnl1999**]** Grassmann Cyprian, Anlauf Joachim K. (1999). *Fast Digital Simulation Of Spiking Neural Networks And Neuromorphic Integration With Spikelab*. International Journal of Neural Systems, Vol. 9, No. 5.

[GraSchWol2002**]** Grassmann Cyprian, Schönauer Tim, Wolff Carsten (2002). *PCNN Neurocomputers – Event Driven and Parallel Architectures*. Proceedings of the European Symposium on Artificial Neural Networks, Brügge, Belgien.

[Gupta1996**]** Gupta A, (1996). *Fast and effective algorithms for graph partitioning and sparse matrix reordering*. IBM Watson Research Center.

[HarFraSch1997**]** Hartmann G., Frank G., Schäfer M., Wolff C., (1997). *SPIKE128K - An Accelerator for Dynamic Simulation of Large Pulse-Coded Networks.* In: Proceedings of the 6th International Conference on Microelectronics for Neural Networks, Evolutionary & Fuzzy Systems, Dresden 1997, S.130-139.

[Haug1986**]** Haug H., (1986). *History of Neuromorphometry*. Journal of Neuroscience Methods, Journal of Neuroscience Methods, Volume 18, S. 1-17.

[Hebb1949**]** Hebb Donald O., (1949). *The Organization of Behavior*. Wiley, New York.

[HenLel1992**]** Hendrickson B., Leland R., (1992). *An improved spectral graph partitioning algorithm for mapping parallel computations*. Sandia National Laboratories.

[HenLel1993**]** Hendrickson B., Leland R., (1993). *A Multilevel Algorithm for Partitioning Graphs*. Sandia National Laboratories.

[Henning1998**]** Henning Michi (1998). *Binding, Migration and Scalability in CORBA*.

[HenVin1999**]** Henning Michi, Vinoski Steve (1999). *Advanced CORBA Programming with C++*, Addison-Wesley.

[HodHux1952**]** Hodgkin A.L., Huxley A.F. (1952). *A Quantitative Description of Membrane Current and its Application to Conduction and Excitation in Nerve*. Journal of Physiology., Vol. 117, S. 500-544

[HubWie1959**]** Hubel D.H., Wiesel T.N. (1959). *Receptive fields of single neurons in the cat's striate cortex*. Journ. Physiology, Vol. 148, S. 574-591

[HubWie1962**]** Hubel D.H., Wiesel T.N. (1962). *Receptive Fields, Binocular Interaction and Functional Architecture in the Cat's Visual Cortex*. Journal of Physiology, Vol. 160, S. 106-154

[HubWie1977**]** Hubel D.H., Wiesel T.N. (1977). *Functional architecture of macaque monkey visual cortex*. Proceedings of the Royal Society of Biological Science, Vol. 198, S. 1-59

[HunGluPal1998**]** Huning H., Glunder H., Palm G., (1998). *Synaptic delay learning in pulse-coupled neurons*. Neural Computation, 10(3):555—565.

[JahLli1984**]** Jahnsen Henrik, Llinas Rodolfo, (1984). *Electrophysiological Properties of Guinea-Pig Thalamic Neurones: An In Vitro Study*. Journal of Physiology, Volume 349, S. 205-226.

[JahRotKla1996**]** Jahnke Axel, Roth Ulrich, Klar Heinrich (1996). *A SIMD / dataflow architecture for a neurocomputer for spike-processing neural networks (NESPINN)*. In: Proceedings of MicroNeuro'96, IEEE Computer Society Press, S. 232-237.

[JanBluhNap1991**]** Jansen, M., Bluhm, M., Napp-Zinn, H., and Eckmiller, R. (1991) *Asynchronous pulse-processing neural net hardware for dynamic functions based on frequency and phase information* In: Proc. 2nd Int. Conf. Microelectronics and Neural Networks, (Ramacher, Rückert, Nossek, eds.), Kyrill & Method - München, pp. 359-365.

[KanSchJes1995**]** Kandel Eric R., Schwartz James H., Jessell Thomas M., (1995). *Neurowissenschaften*. Spektrum Akademischer Verlag.

[KarKum1998**]** Karypis G., Kumar V., (1998). *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs*. Department of Computer Science, University Minnesota.

[KarKum1998a**]** Karypis G., Kumar V., (1998). *Multilevel k-way Partitioning Scheme for Irregular Graphs*. Department of Computer Science, University Minnesota.

[KarKum1998b**]** Karypis G., Kumar V., (1998). *Multilevel Algorithms for Multi-Constraint Graph Partitioning*. Department of Computer Science, University Minnesota.

[KerLin1970**]** Kernighan B.W., Lin S., (1970). *An efficient heuristic procedure for partitioning graphs*. The Bell System Technical Journal, 49(2), S. 291-307.

[KrüAip1988**]** Krüger J., Aiple F., (1988). *Multimicroelectrode Investigation of Monkey Striate Cortex: Spike Train Correlations in the Infragranular Layers*. Journal of Neurophysiology, Volume 60, S. 799-829.

[LanAda1986**]** Lancaster B., Adams P.R., (1986). *Calcium-Dependent Current Generating the Afterhyperpolarization of Hippocampal Neurons*. Journal of Neurophysiology, Volume 6, S. 1268-1282.

[LawKel1982**]** Law Averill M., Kelton W. David, (1982). *Simulation Modeling and Analysis*. McGraw-Hill.

[LeeTep1997**]** Lee R. C., Tepfenhart W. M., (1997). *UML and C++ - A practical guide to object-oriented development*. Prentice Hall, Upper Saddle.

[Llinas1988**]** Llinas Rodolfo, (1988). *The Intrinsic Electrophysiological Properties of Mammalian Neurons: Insights Into Central Nervous System Function*. Science, Volume 242, S. 1654-1664.

[LliSug1980**]** Llinas R., Sugimori M., (1980). *Electrophysiology of Mammalian inferior olivary neurons in vitro. Different types of voltage dependant ionic conductances*. Journal of Physiology (London), Volume 315, S. 549-567.

[Lohwasser1999**]** Lohwasser Christoph, (1999). *GUI (SPIKELAB)*, Diplomarbeit, Institut für Informatik – Technische Informatik, Universität Bonn.

[LoRidGri2000**]** Lo Sai-Lai, Riddoch D., Grisby , (2000). *The omniORB2 version 3.0 User's Guide*. AT&T Laboratories, Cambridge**.**

[Luksch1993**]** Luksch P., (1993). *Parallelisierung ereignisgetriebener Simulationsverfahren auf Mehrprozessorsystemen mit verteiltem Speicher*. Dissertation, Technische Universität München, Verlag Dr. Kovač, Hamburg.

[Maass1995**]** Maass W., (1995). *On the computational complexity of networks of spiking neurons*. Advances in Neural Information Processing Systems, vol.7, MIT Press: S. 183-190.

[Maass1996**]** Maass Wolfgang, (1996). *Lower bounds for the computational power of spiking neurons*. Neural Computation, 8: S. 1-40.

[Maass1997**]** Maass Wolfgang, (1997). *On the relevance of time in neural computation and learning*. In Proceedings of the 8th International Workshop on Algorithmic Learning Theory ALT'97, eds. M. Li and A. Maruoka, Lecture Notes in Artificial Intelligence, vol. 1316, S. 364–384. Springer, Berlin.

[MaaBis1998**]** Maass Wolfgang, Bishop Christopher M., (1998). *Pulsed Neural Networks*, MIT Press.

[Malsburg1981**]** Malsburg von der, C., (1981). *The correlation theory of brain function*. Interner Bericht 81-2, Max Planck Institut für Biophysikalische Chemie.

[Malsburg1987**]** Malsburg von der, C., (1987). *Synaptic plasticity as basis of brain organization*. In J.-P. Changeux and M. Konishi, editors, *The Neural and Molecular Bases of Learning*, Dahlem Konferenzen, S. 411-431. John Wiley & Sons Ltd., Chichester.

[Marin1997**]** Marín Mauricio, (1997). *An Emperical Comparison of Priority Queue Algorithms*, Technical report PRG-TR-10-97, Oxford University.

[MarLueFro1997**]** Markram H., Lübke J., Frotscher M., Sakmann B., (1997). *Regulation of Synaptic Efficacy by Coincedence of Postsynaptic APs and EPSPs*. Science, Volume 275, S. 213-215.

[McCConLig1985**]** McCormick David A., Connors Barry W., Lighthall James W., Prince David A., (1985). *Comparative Electrophysiology of Pyramidial and Sparsely Spiny Stellate Neurons of the Neocortex*. Journal of Neurophysiology, Volume 54, S. 782-806.

[Meissner2000**]** Meissner Michael, (2000). *Einsatz von CORBA im verteilten Simulator für pulsverarbeitende Neuronale Netze (SPIKELAB)*, Diplomarbeit, Institut für Informatik – Technische Informatik, Universität Bonn.

[Meyers1995**]** Meyers S., (1995). *Effektiv C++ programmieren*. 2. Auflage, Addison-Wesley, Bonn.

[OMG1999**]** OMG (1999). *The Common Object Request Broker: Architecture and Specification.* Rev. 2.3, Object Management Group.

[RamBeiBrü1991**]** Ramacher Ulrich, Beichter J., Brüls Nikolaus (1991). *Architecture of a General Purpose Neural Signal Processor*. In: Proceedings of the International Joint Conference on Neural Networks (IJCNN'91), S.443-446.

[Richardson2000a**]** Richardson Tristan (2000). *The OMNI Thread Abstraction*. Olivetti & Oracle Research Laboratory, Cambridge.

[Richardson2000b**]** Richardson Tristan, (2000). *The OMNI Naming Service*. Olivetti & Oracle Research Laboratory, Cambridge.

[RieWarRuy1997**]** Rieke Fred, Warland David, de Ruyter van Steveninck Rob, Bialek William, (1997). *Spikes: Exploring the neural code*. Serie: Sejnowski Terrence J., Poggio Tomaso A. (eds.), Computational Neuroscience, MIT Press.

[RinDotDem1994**]** Ringo James L., Doty Robert W., Demeter Steven, Simard Patrice Y., (1994). *Time Is Of The Essence: A Conjecture That Hemispheric Specialization Arises from Interhemispheric Conduction Delay*. Celebral Cortex, Volume 4, S. 331-343.

[RönAya1997**]** Rönngren R., Ayani R., (1997). *A Comparative study of sequential and parallel priority queue algorithms*. In ACM transactions on modeling and computer simulation, Vol 7(2): 157 - 209.

[Rolls1989**]** Rolls E.T., (1989). *The representation and storage of information in neuronal networks in the primate cerebral cortex and hippocampus*. The Computing Neuron, Addison-Wesley: S. 125-159.

[Sauermann2001**]** Sauermann Mirko, (2001). *Entwicklung eines Gerätetreibers für Windows NT zur Ansteuerung eines PCI-Adapters für parallele Datenflußmodule*, Diplomarbeit, Institut für Informatik – Technische Informatik, Universität Bonn.

[SchHar1999**]** Schäfer Martin, Hartmann Georg (1999). *A Flexible Hardware Architecture for Online Hebbian Learning in the Sender-Oriented Neurocomputer Spike 128K*. In: Proceedings of the 7th International Conference on Microelectronics for Neural, Fuzzy and Bio-Inspired Systems, S. 316-323, Granada, Spanien.

[Schäfer2000**]** Schäfer Martin, (2000). *Lernen in Neurocomputern für große pulscodierte neuronale Netze*. Dissertation, Fachbereich 14 Elektrotechnik und Informationstechnik, Universität-GH Paderborn.

[Schild1999**]** Schild Wolfgang, (1999). *Entwicklung und Implementierung eines Nachrichtensystems für die verteilte eventgetriebene Simulation Neuronaler Netze*, Diplomarbeit, Institut für Informatik – Technische Informatik, Universität Bonn.

[Schmidt1987**]** Schmidt Robert F., (1987). *Grundriß der Neurophysiologie*. 6. Auflage Springer Verlag.

[Schmitt1999**]** Schmitt Michael, (1999). *On the implications of delay adaptability for learning in pulsed neural networks*. In Demiris, J. und Westermann G. (eds.), Proceedings of the Workshop on Biologically-Inspired Machine Learning, S. 28–37. Chania, Greece.

[SchAtaMeh2000**]** Schönauer Tim, Atasoy S., Mehrtash Nasser, Klar Heinrich (2000). *Simulation of a Digital Neuro-Chip for Spiking Neural Networks*. In: Proceedings of the IEEE International Joint Conference on Neural Networks, IJCNN2000, July, Como, Italy.

[SchMehJah1998**]** Schönauer Tim, Mehrtash Nasser, Jahnke Axel (1998). *Novel Concepts for a Neuro-Accelerator for Spiking Neural Networks*. In: Workshop on Virtual Intelligence and Dynamic Neural Networks – VIDYNN'98, Stockholm.

[SchMehKla1998**]** Schönauer Tim, Mehrtash Nasser, Klar Heinrich (1998). *Architecture of a Neuroprocessor Chip for Pulse-Coded Neural Networks*. In: Proceedings of International on Computer Intelligence and Neurons – ICCIN'98, JCIS'98, Research Triangle Park, S. 17-20, NC (USA).

[Schüz1995**]** Schüz Almut, (1995). *Neuroanatomy in a Computational Perspective*. In Arbib, Michael (ed.), The Handbook of Brain Theory and Neural Networks, S. 622-626, MIT Press.

[Schulz1999**]** Schulz S., (1999). *Implementierung einer Graphenpartitionierung für die verteilte Simulation pulsverarbeitender neuronaler Netze*. Diplomarbeit, Universität Bonn.

[SenTsoMar1997**]** Senn W., Tsodyks M., Markram H., (1997). *An algorithm for synaptic modification based on precise timing of pre and postsynaptic action potentials*, Lecture Notes in Computational Science, S. 121-126.

[Spektrum1988**]** Spektrum der Wissenschaft, (1988). *Gehirn und Nervensystem*. Spektrum-Verlag.

[SteWil1997**]** Steuber V., and Willshaw D. J., (1997). *How a single purkinje cell could learn the adaptive timing of the classically conditioned eye-blink response*. In Proceedings of the Seventh International Conference on Artificial Neural Networks: ICANN-97, S. 115–120. Berlin; New York: Springer-Verlag.

[StoReiEck1996**]** Stoecker, M., Reitböck, J.R., Eckhorn, R. (1996). *A Neural Network for Scene Segmentation by Temporal Coding.* Neurocomputing, Vol. 11, pp. 123-134.

[Thompson1994**]** Thompson Richard E., (1994). *Das Gehirn: Von der Nervenzelle zur Verhaltenssteuerung*. Spektrum Akademischer Verlag.

[Troidl2000**]** Troidl S., (2000). *Generierung und Visualisierung räumlich und hierarchisch strukturierter neuronaler Netze*. Diplomarbeit, Universiät Bonn.

[TsiLipMad1988**]** Tsien R.W., Lipscombe D., Madison D.V., Bley K.R., Fox A.P., (1988). *Multiple types of calcium channels and their selective modulation*. Trends in Neuroscience, Volume 11, S.431-438.

[Weber1999**]** Weber Karsten, (1999). *Ereignisgetriebene Simulation pulsverarbeitender Neuronenmodelle*. Diplomarbeit, Universität Bonn.

[Werkhausen2002**]** Werkhausen Axel, (2002). *Effiziente Ereignislistenverwaltung in einem Simulator für pulsverarbeitende neuronale Netze*. Diplomarbeit, Universität Bonn.

[Wolff2001**]** Wolff Carsten, (2001). *Parallele Simulation großer pulscodierter neuronaler Netze*. Dissertation, Fachbereich 14 Elektrotechnik und Informationstechnik, Universität-GH Paderborn.

[WonPri1981**]** Wong R.K.S., Prince D.A., (1981). *Afterpotential Generation in Hippocampal Pyramial Cells*. Journal of Neurophysiology, Volume 45, S. 86-97.

[Watts1994**]** Watts Lloyd, (1994). *Event-Driven Simulation of Networks of Spiking Neurons*. Advances in Neural Information Processing Systems 6 (NIPS-6 '93). Jack D. Cowan and Gerald Tesauro and Joshua Alspector, S. 927-934, Morgan Kaufmann.

[Zell1994**]** Zell Andreas, (1994). *Simulation neuronaler Netze*. Addison-Wesley.

Kapitel 10 - SPIKELAB Bedienung und Programmierung

10.1. Graphische Oberfläche

In Abschnitt 4.1.1 wurden bereits die wesentlichen Merkmale der graphischen Oberfläche beschrieben. Kern der Oberfläche stellt eine Visualisierung dar, welche die Hierarchie der zu simulierenden Netzwerke in einer Baumstruktur wiederspiegelt. Die Visualisierung der Baumstruktur entspricht der des Windows Explorer. Mit dieser Analogie können Neuronen als Dateien aufgefasst werden, welche in Ordnern, die Schichten oder Pools entsprechen, zusammengefasst werden. Verknüpfungen existieren immer nur zwischen Neuronen, d.h. dass Ordner alleine nur der hierarchischen Gliederung und Visualisierung dienen. Im Folgenden werden im Sinne einer schrittweisen Anleitung die Funktionen der Oberfläche detaillierter dargestellt.

10.1.1. Start des Programms

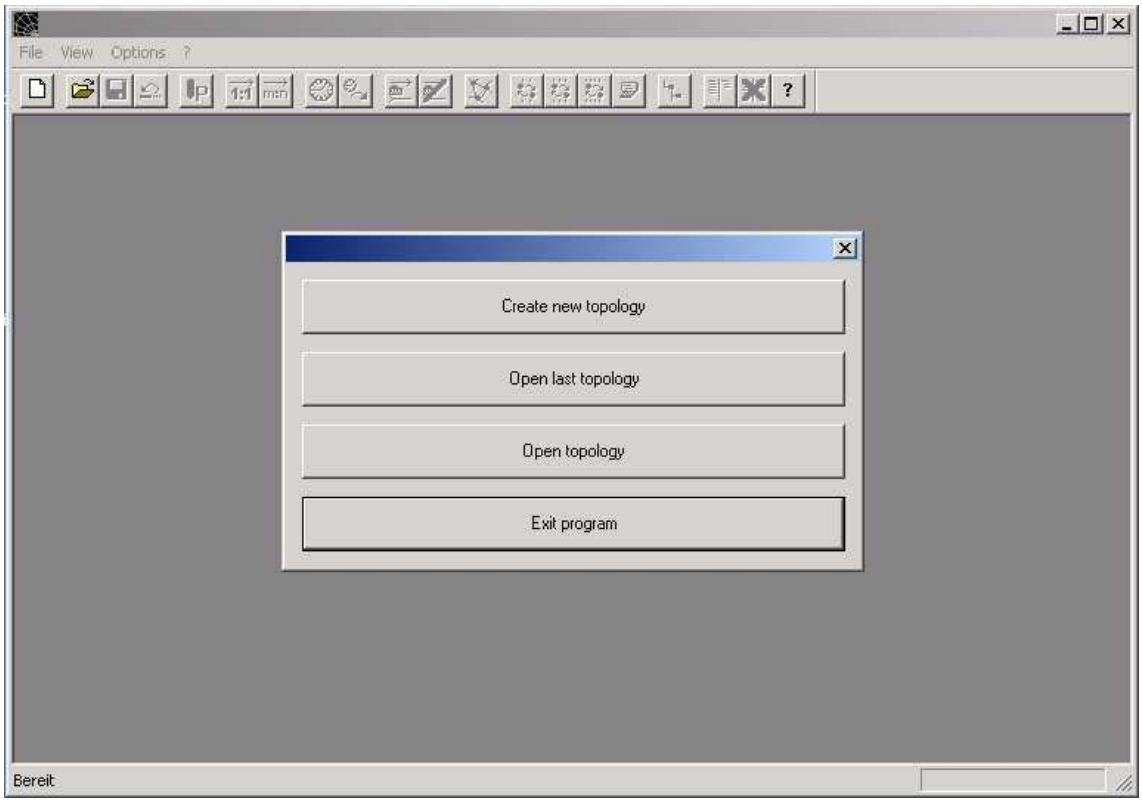


Abbildung 77: Startdialog von Spikelab

Nach dem Start des Programms Spikelab.exe erscheint der in Abbildung 77 dargestellte Startdialog, welcher vier Möglichkeiten zur Wahl stellt:

1. Eine leere Topologie zu erstellen (Create new topology)
2. Die zuletzt geöffnete Topologie zu öffnen (Open last topology)
3. Eine beliebige Topologie zu öffnen (Open topology)
4. Das Programm zu beenden (Exit program)

Beim Öffnen wie auch beim Speichern von Topologien erscheint ein Windows-Standard-Dialog, der die Auswahl einer Quell- bzw. Zieldatei erlaubt.

Wird die erste Option gewählt, so wird eine leere Topologie erstellt (Abbildung 78) und es wird die dazu gehörige Ansicht erstellt. Die Ansicht enthält einen Knoten, die Wurzel (root), an welcher alle weiteren Elemente in der Topologie angehangen werden.

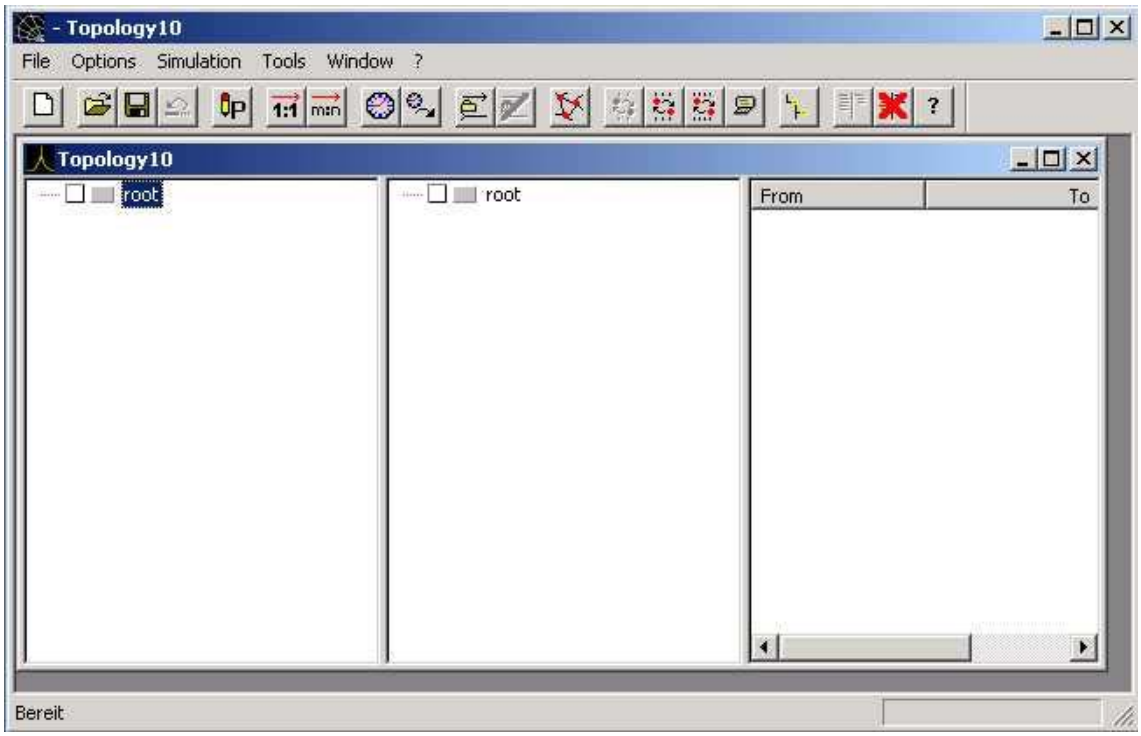


Abbildung 78: Dokumentenfenster einer leeren Topologie (hier: Topology10)

10.1.1.1. Manuelle Eingabe eines Netzwerks

Mit der rechten Maustaste kann das Kontextmenü auf dem Wurzelknoten aufgerufen werden. Über dieses Kontextmenü können Neuronen oder Ordner hinzugefügt, gelöscht, deren Einstellungen geändert und die Aufzeichnung von Parametern während der Simulation (Logging) aktiviert bzw. deaktiviert werden. In Abbildung 79 wurde das Kontextmenü aufgerufen und der Eintrag zum Hinzufügen von Neuronen ist hervorgehoben. Wird diese Auswahl aktiviert, so erscheint der in Abbildung 80 dargestellte Dialog, mit dem die Anzahl der einzufügenden Neuronen angegeben, ein Name für das Neuron und über eine Auswahlliste ein Simulationsmodell ausgewählt werden kann. Außerdem gibt der Dialog mit der „*Insert position*“ an, an welcher Stelle die Neuronen eingefügt werden. Die „*Insert position*“ entspricht dabei der Selektion, von der aus das Kontextmenü aufgerufen wurde. Im dargestellten Dialog werden daher 5 Neuronen am Wurzelknoten eingefügt, wobei das Simulationsmodell „StdModIAFf“ hinterlegt wird und die Neuronen jeweils als InputNeuron benannt werden.

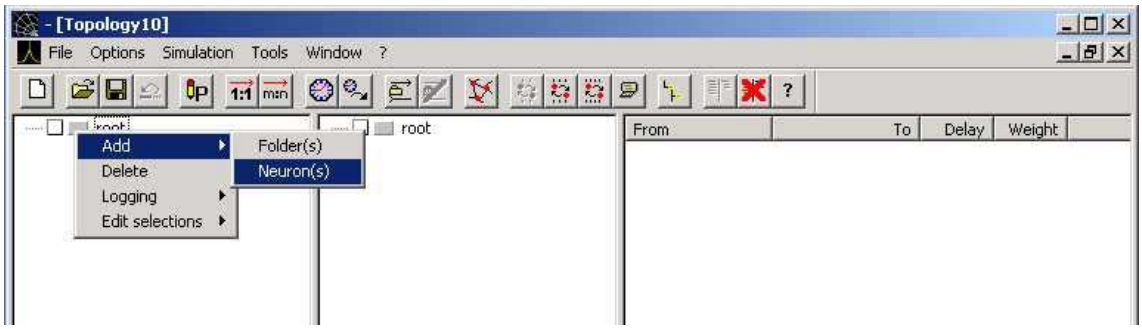


Abbildung 79: Kontextmenü zum Hinzufügen, Entfernen und Modifizieren von Neuronen und Ordnern

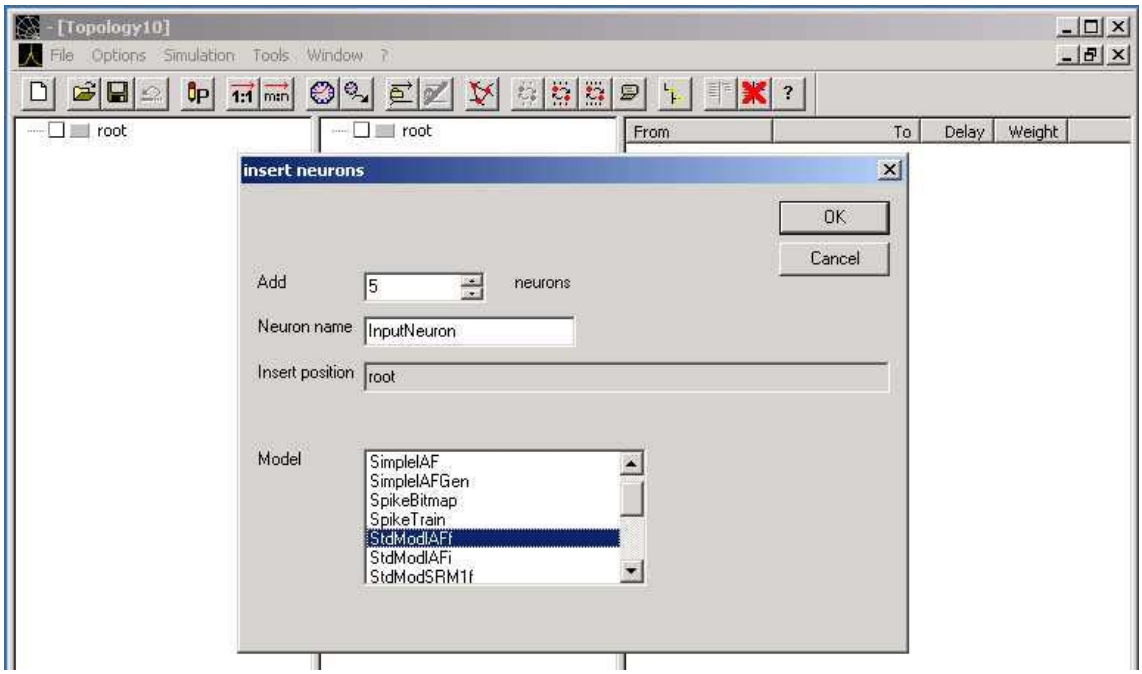


Abbildung 80: Dialog zum Hinzufügen von Neuronen.

Werden die Angaben mit „OK“ übernommen so wird die Topologieansicht aktualisiert und die Neuronen unterhalb des Wurzelknotens angezeigt. Werden die Neuronen durch die Aktivierung des Pluszeichens vor dem Neuron weiter expandiert, so werden die für das betreffende Modell verfügbaren Eingangs und Ausgangstypen sichtbar. In Abbildung 81 sind alle Neuronen auf diese Weise expandiert worden.

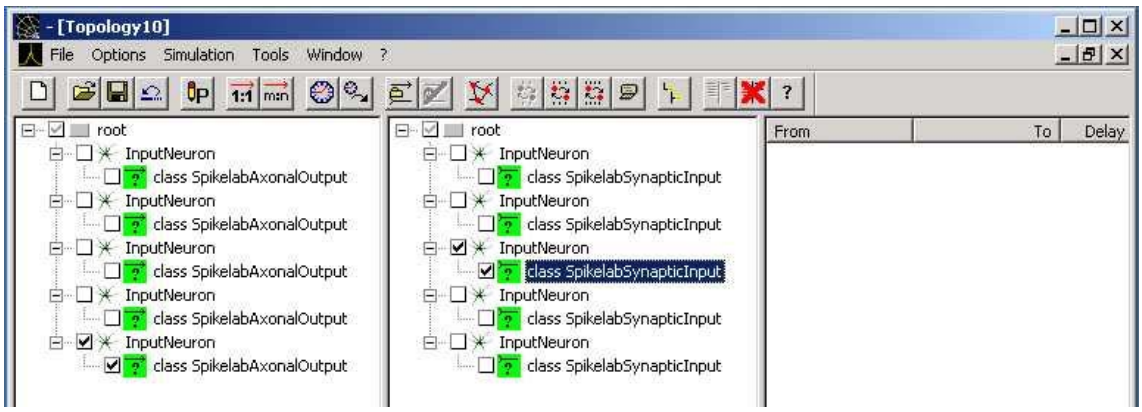


Abbildung 81: Eingangs- und Ausgangstypen von Neuronen

Die Baumstruktur ist doppelt ausgeführt, um die Selektion von Neuronen für eine Vernetzung intuitiv zu gestalten. Neuronen, die im linken Baum selektiert werden, dienen als Ausgangspunkt der Vernetzung und Neuronen, die im rechten Baum selektiert dienen als Endpunkt. Durch die Selektion des Neurons wird immer auch der im Modell angegebene Standardeingang bzw. Standardausgang mit selektiert. Sind mehrere Ausgangs- oder Eingangstypen vorhanden, können durch die beschriebene Expansion gezielt die entsprechenden Typen für eine Verknüpfung angewählt werden. Durch die Tasten „1:1“ bzw. „n:m“ aus der Werkzeugleiste können die so selektierten Neuronen verknüpft werden. Hierbei führt die Wahl „1:1“ zu einer Verknüpfung eines auf der linken Seite selektierten Neurons mit genau einem auf der rechten Seite selektierten Neuron. Es wird dabei die selektierte Menge von oben nach unten verarbeitet. Die Wahl „n:m“ führt hingegen dazu, das jedes Neuron, das auf der linken Seite selektiert ist mit jedem auf der rechten Seite selektierten Neuron verknüpft wird.

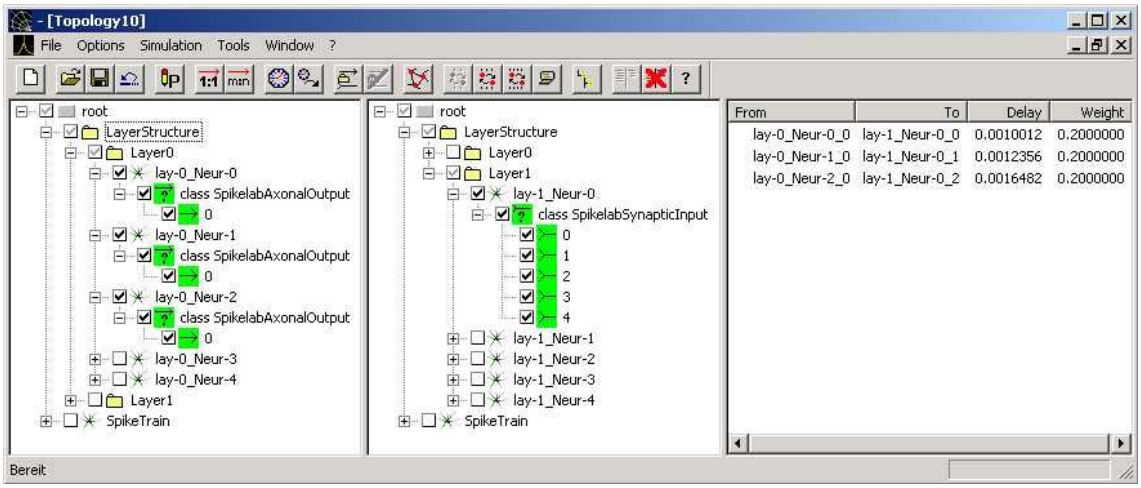


Abbildung 82: Visualisierung konkreter Eingänge und der Verknüpfungen zwischen Eingängen und Ausgängen

In der Regel werden erst beim Erstellen von Verknüpfungen konkrete Eingänge bzw. Ausgänge angelegt. Dies ist insbesondere für Eingänge recht sinnvoll, da meist die Zahl der Eingänge an einem Neuron von der Struktur des Netzwerkes abhängt. In Abbildung 82 ist eine zweischichtige, vollvernetzte Struktur dargestellt. Auf der linken Seite sind die ersten drei Neuronen der ersten Schicht vollständig expandiert und selektiert. Auf der rechten Seite ist das erste Neuron der zweiten Schicht vollständig expandiert und selektiert. Durch die Vollvernetzung besitzt das Neuron aus der zweiten Schicht fünf konkrete Eingänge (0 bis 4) unterhalb des betreffenden Eingangstypen (SpikelabSynapticInput). Drei dieser Eingänge werden von den drei auf der rechten Seite selektierten Neuronen der ersten Schicht gespeist. Die dazu gehörigen Verbindungen sind in der Liste rechts neben den beiden Baumstrukturen aufgeführt. Die Verbindungen zu den Eingängen 3 und 4 des Neurons lay-1_Neur-0 werden nicht aufgelistet, obwohl die Eingänge selektiert sind, da die speisenden Neuronen lay-0_Neur-3 und lay-0_Neur-3, respektive deren Ausgänge, auf der rechten Seite nicht selektiert sind.

10.2. Netzwerkgeneratoren

Netzwerkgeneratoren vereinfachen die Erstellung regulärer Netzwerkstrukturen erheblich. Zum Abschluss dieser Arbeit liegen Netzwerkgeneratoren für die Generierung vollvernetzter Netzwerke (HopfieldGenerator), für geschichtete Netzwerke

(LayerNetGenerator), für Poolstrukturen (PoolGenerator) und eine spezielle Netzwerkstruktur für die Verarbeitung von Bewegtbildsequenzen (RSTNetGenerator) vor.

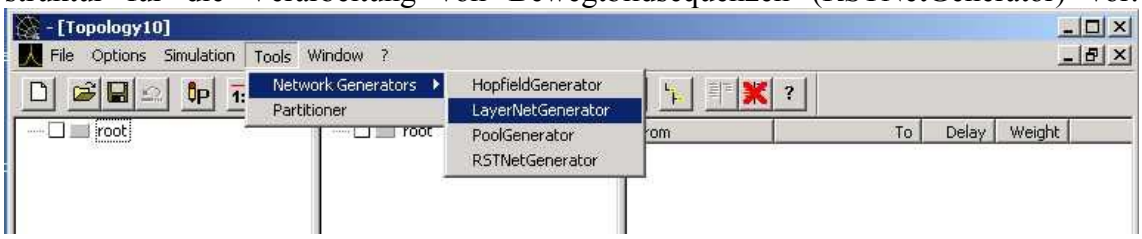
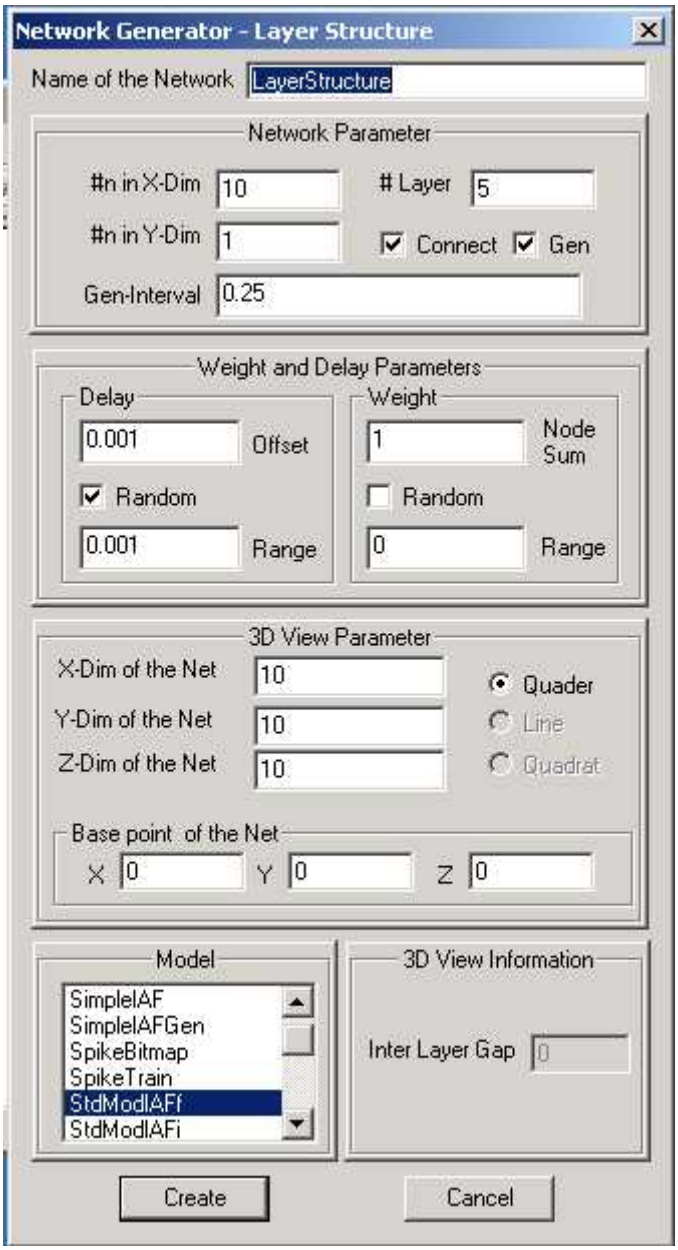


Abbildung 83: Der Aufruf eines Netzwerkgenerators erfolgt über das Menü „Tools“→“Network Generators“



Zusätzliche Netzwerkgeneratoren können über eine entsprechende API in Spikelab integriert werden und stehen nach einem erneuten Linkvorgang der Gesamtapplikation umgehend zur Verfügung. Eine Beschreibung der API, welche bei der Implementierung eines Netzwerkgenerators einzuhalten ist, findet sich in Abschnitt 10.7.

Abbildung 84 zeigt den Dialog des Netzwerkgenerators für geschichtete Netzwerke. Im oberen Bereich des Dialogs können die Netzwerkparameter, wie die Ausdehnung der Schichten in X- und Y-Richtung und die Gesamtzahl der Schichten gewählt werden. Zusätzlich kann dort gewählt werden, ob die Schichten vernetzt werden sollen, und ob die erste Schicht durch ein Generatorneuron gespeist werden soll, als auch das Intervall mit dem dieses Generatorneuron Pulse erzeugt.

Im Abschnitt darunter können die Vorgaben für die Einstellung der Verzögerungen und Gewichte vorgenommen werden. Für die Verzögerung kann ein Offset und ein Bereich angegeben werden,

Abbildung 84: Dialog des Netzwerkgenerators für geschichtete Netzwerke

in dem die Verzögerungswerte oberhalb des Offsets mit zufälligen Werten initialisiert werden sollen. Wird keine Randomisierung der Verzögerungen gewählt, so werden alle Verzögerungswerte auf den Wert des Offsets festgelegt. Für die Gewichte der Verbindungen kann entweder ein Summengewicht für alle Eingänge eines Neurons

angegeben werden, welches dann unter den zu erzeugenden Eingängen gleich verteilt wird, oder es erfolgt eine Randomisierung wie bei den Verzögerungen.

Im Abschnitt unterhalb der Gewichts- und Verzögerungseinstellungen können die Visualisierungsparameter für die 3-D Darstellung des Netzwerks eingestellt werden. Dies sind die räumlichen Ausdehnungen des Netzwerks und der Bezugspunkt im dreidimensionalen Raum.

Ganz unten im Dialog kann schließlich das Neuronenmodell gewählt werden, welches bei der Generierung verwendet werden soll.

Die Dialoge der anderen Netzwerkgeneratoren sind anlog aufgebaut und unterscheiden sich lediglich im Hinblick auf die für die Netzwerkstruktur spezifischen Parameter.

10.3. Simulation

Zur Simulation eines Netzwerks muss der Simulationsdialog über den Menüpunkt Simulation→Status aufgerufen werden (Abbildung 85). Über den Menüpunkt Simulation→Logging lassen sich für das gesamte Netzwerk Einstellungen zur Aufzeichnung von Daten während der Simulation vornehmen.

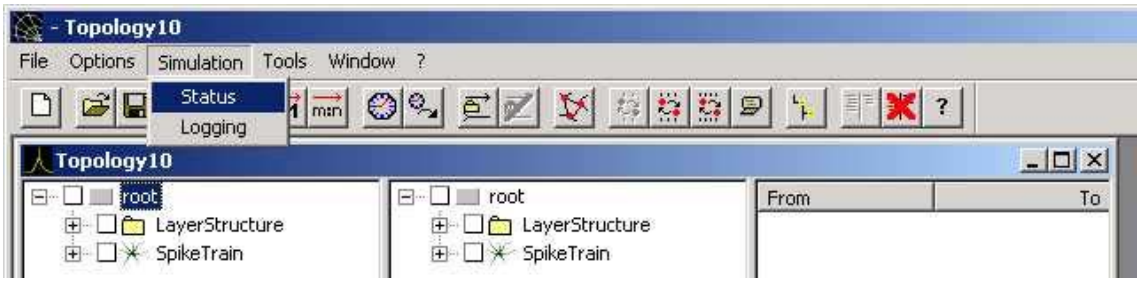


Abbildung 85: Aufruf des Simulationsdialogs über das Menü Simulation→Status

Der Simulationsdialog (Abbildung 86) gestattet es ein Simulationsprojekt über „*Browse*“ zu laden, oder das aktuelle Projekt aus der graphischen Oberfläche für die Simulation zu verwenden. In jedem Falle muss der Simulator initialisiert werden. Dies geschieht durch „*Initialize*“. „*Reset*“ wird verwendet um den Simulator auf die Simulationszeit Null zurückzusetzen und für das geladene Netzwerk ein Reset aufzurufen. „*Restart*“ entfernt ein eventuell zuvor geladenes Netzwerk und setzt den Simulator auf die Simulationszeit Null zurück.

Nach der Initialisierung ändert sich der Simulationsstatus von „*UNKNOWN*“ auf „*STOPPED*“ und die Simulation kann über „*Start*“ gestartet, mit „*Stop*“ angehalten und mit „*Continue*“ wieder fortgesetzt werden. Die Simulationsdauer kann relativ zum aktuellen Zeitpunkt „*Simulate for*“ oder absolut „*Simulate until*“ angegeben werden.

Wurde der Simulator gestartet, wechselt der Simulationsstatus auf „*RUNNING*“ und der Simulationsfortschritt kann in dem Feld „*Simulation Status*“ verfolgt werden. Für eine Einzelplatzsimulation wird die aktuelle Simulationszeit angezeigt. Bei einer verteilten Simulation wird neben der aktuellen Simulationszeit auch die Startzeit und Endzeit des Simulationsrechners angezeigt, auf dem die Simulationskontrolle ausgeführt wird. Zudem wird in den Feldern „*Virtual fastest*“ und „*Virtual slowest*“ die Simulationszeit des am weitesten und des am wenigsten weit fortgeschritten Simulationsrechners angezeigt.

Die virtuelle Zeit gibt den simulierten Zeitraum an und die reale Zeit die seit dem Start der Simulation tatsächlich verstrichene Zeit. Darunter wird die Zahl der insgesamt verarbeiteten Ereignisse und als Teilmenge hiervon die Gesamtzahl der verarbeiteten Pulse, sowie jeweils rechts daneben, die verarbeiteten Ereignisse bzw. Pulse pro tatsächlich verstrichener Sekunde angezeigt.

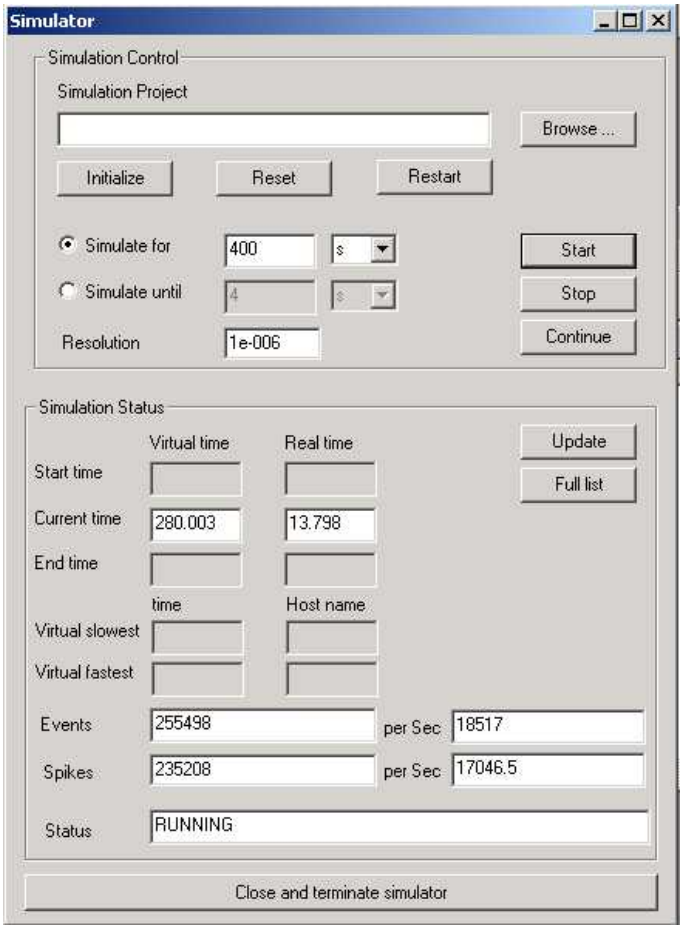


Abbildung 86: Simulationsdialog

10.4. Kommandozeilenparameter für den Programmstart

Kommandozeilenparameter für die graphische Umgebung können dazu eingesetzt werden mehrere Simulationsläufe unbeaufsichtigt hintereinander auszuführen. Dabei können die Netzwerkgeneratoren mit unterschiedlichen Parametern aufgerufen werden.

Eingeleitet wird der so genannte „*Batch*"-Betrieb durch den Kommandozeilenparameter –b. Folgende Kommandozeilenparameter stehen im Zusammenhang mit dem Parameter –b zur Verfügung.:

-ngt [Wert] „*(n)etwork (g)enerator (t)ype*" gibt den zu verwendenden Netzwerkgenerator an. Unterstützt wird 0 = geschichtete Netzwerke und 1 = vollvernetzes Netzwerk.
-n [Wert] „*(n)umber of layers*" für die Anzahl zu erzeugenden Netzwerkschichten
-r [Wert] „*(r)esolution*" für die Auflösung mit der die Simulation durchgeführt werden soll.
-s [Wert] „*(s)ize of layer*" für die Größe der zu generierenden Layer
-m [Wert] „(m)odel id" für die Identifikationsnummer des zu verwendenden Neuronenmodells

-si [Wert] „(s)pike train (i)nterval" für das Intervall mit dem das speisende Generatorneuron Pulse erzeugen soll.
-t [Wert] „*(t)ime*" für die Dauer des zu simulierenden Zeitintervalls
-do [Wert] „(d)elay (o)ffset" für den Offset der Verzögerung
-dr [Wert] „(d)elay (r)ange" für den Bereich in dem die Verzögerung randomisiert werden soll
-thr [Wert] „(thr)eshold" für den Schwellenwert des Neuronenmodells

Als Beispiel sei ein Eintrag für eine Stapelverarbeitungsdatei angegeben, deren Ausführung zur Simulation von geschichteten Netzwerken mit jeweils 11 Schichten und einer Schichtbreite von 10 bis 200 Neuronen in 10'er Schritten führt. Jede Netzwerkgröße wird dabei mit unterschiedlichen Aktivitäten simuliert, indem in einer äußeren Schleife das Intervall des Generatorneurons von 0,0625 bis 16 variiert wird:

```
for %%r in (0.000001) do
   for %%i in (0.0625 0.125 0.25 0.5 1 2 4 8 16) do
      for /L %%s in (10,10,200) do
           Spikelab -b -n 11 -t 10 -s %%s -si %%i -r %%r
```

10.5. Betrieb über die Konsole – Fileformat

Der Simulator kann auch ohne die graphische Oberfläche verwendet werden. Diese Version (Projekt SubSimulatorKonsole) liest Netzwerke aus drei Dateien ein. Die Partitionsdatei beschreibt die Struktur des Netzwerks und die darin enthaltenen Verzögerungen und besitzt die Dateiendung „prt". Die Gewichtsdatei beinhaltet alle Parameter der verwendeten Neuronenmodelle und die Gewichte der Verbindungen, welche Eingangsparameter zu den Neuronen sind. Die Gewichtsdatei trägt die Endung „wgt". Die Log-Datei beschreibt welche Neuronendaten während der Simulation aufgezeichnet werden sollen und sie besitzt die Dateiendung „lgi".

Bei der verteilten Simulation kommt noch die „Hostdatei" mit der Endung „hst" hinzu. In dieser Datei werden die Zuordnungen zwischen Rechnern und den darauf zu simulierenden Partitionen angegeben.

10.5.1. Partitionsdatei

Die Partitionsdatei besitzt folgendes Format (#Kommentare):

```
SimpleTest  #Name des Netzwerks

4           #Zahl aller logischen Prozesse in dem folgenden Netzwerk
$OPTIONS    #Schlüsselwort leitet den Abschnitt für Optionen ein
floatDelay = TRUE #Diese, einzig wesentliche Option gibt an, dass die
                  #Darstellung der Verzögerungen in Fließkommazahlen
                  #erfolgt

$PROCESSES  #Schlüsselwort leitet den Abschnitt ein, in dem normale
            #logische Prozesse – keine Generatoren beschrieben und
            #verknüpft werden.

P1 (4)                  #Name des Prozesses und in Klammern die Nummer
                        #des Neuroenenmodells
local: P2(0.001,0)      #local: leitet die lokalen Verbindungen ein,
                        #welche der Form NameZielprozess(Ausgang,
                        #Verzögerung, EingangZielprozess) folgen, wobei
```

```
                        #der Ausgang weggelassen werden kann.
extern:                 #extern: leitet die externen Verbindungen ein.
                        #Hinter diesem Schlüsselwort werden die Namen
                        #aller Partitionen aufgeführt zu denen der
                        #Prozess eine Verbindung hat
;                       #Das Semikolon schließt die Beschreibung eines
                        #logischen Prozesses ab.

P2 (4)
local: P3(0.001,0)
extern:
;

P3 (4)
local:
extern:
;

$GENERATORS             #Schlüsselwort leitet den Abschnitt der
                        #Generatorneuronen ein.
                        #Die Beschreibung der Prozesse erfolgt wie
                        #bei den normalen logischen Prozessen

P4 (6)
local: P1(0.001,0)
extern:
;

$PARTITIONS             #Schlüsselwort leitet den Abschnitt ein, in dem
                        #die eingehenden Verbindungen von anderen
                        #Partitionen aufgeführt werden. (siehe unten)

END_MARKER              #Schließt die Datei ab.
```

Ein Beispiel für einen $PARTITIONS Abschnitt:

Jede externe Partition, die Verbindungen in die aktuelle Partition besitzt, wird separat gelistet, und darunter für jeden Prozess aus der externen Partition ein so genannter „*connector*“ eingerichtet, der mit dem Schlüsselwort *connector* eingeleitet wird. Danach folgen die Prozessnamen der lokalen Zielprozesse gefolgt von der Verzögerung und dem betreffenden Eingang des lokalen Prozesses.

```
$PARTITIONS
part-3:
connector P17: P1(0.001546,1), P2(0.001175,1), P3(0.001484,1)
connector P18: P1(0.001189,2), P2(0.001379,2), P3(0.001251,2)
part-4:
connector P20: P1(0.001546,3), P2(0.001175,3), P3(0.001484,3)
connector P21: P1(0.001189,4), P2(0.001379,4), P3(0.001251,4)
```

10.5.2. Gewichtsdatei

Der Aufbau der Gewichtsdatei ist sehr einfach gehalten und wird im Wesentlichen durch die Strukturen bestimmt, in denen die Modelle Ihre Parameter ablegen. In einer Gewichtsdatei werden daher nicht nur Gewichte, sondern jegliche Parameter der logischen Prozesse abgelegt.

Beispiel für eine Gewichtsdatei:

```
$PROCESS$               # Jeder Prozess wird mit diesem Schlüsselwort
                        # eingeleitet
P1                      # Name des Prozesses – muss zur *.prt Datei
                        # korrespondieren
$DO_RESET$ 1            # Schlüsselwörter und Daten des Modells
$ABS_REFRAC$ 0.001      # entsprechend den Vorgaben des Modell-
$THRESHOLD$ 1.0         # schreibers.
$WEIGHTS$ 1
110 0.01 1000

$PROCESS$
P2
$DO_RESET$ 1
$ABS_REFRAC$ 0.001
$THRESHOLD$ 1
$WEIGHTS$ 1
110 0.01 1000

$PROCESS$
P3
$DO_RESET$ 1
$ABS_REFRAC$ 0.001
$THRESHOLD$ 1
$WEIGHTS$ 1
110 0.01 1000
$PROCESS$ P4
$INTERVAL$ 0.1
$END_MARKER$            # Abschluss der Datei
```

10.5.3. Log-Datei

Mit Hilfe der Log-Datei kann die Aufzeichnung aller Pulse und Statusvariablen über einen gewählten Zeitraum aktiviert werden. Darüber hinaus kann für einzelne Prozesse die Aufzeichnung selektiv für entsprechende Zeiträume aktiviert oder deaktiviert werden.

Sofern die Aufzeichnung aller Pulse, respektive Statusvariablen aktiviert ist, so wirkt die Nennung eines spezifischen Prozesses als Deaktivierung für den angegebenen Zeitraum. Ist die Aufzeichnung für alle Pulse bzw. Statusvariablen deaktiviert, so wirkt die Nennung als Aktivierung für den angegebenen Zeitraum.

Beispiel für eine Log-Datei:

```
ALL_SPIKES 0 INF        # Standardeintrag um alle Pulse im Netz
                        # aufzuzeichnen (Hier beginnend mit dem Zeit-
                        # punkt 0 bis zur maximalen Simulationsdauer
                        # INF = infinity

ALL_MODELS 0 INF        # Standardeintrag um alle Statusdaten im Netz
                        # aufzuzeichnen

LOG_GRANULARITY 10      # Granularität mit der Statusvariablen
                        # aufgezeichnet werden sollen

PROCESS P1 EVENTS 0.001 5     # Einschalten bzw. Ausschalten der
                              # Pulsaufzeichnung für einen spezifischen
                              # Prozess für den angegebenen Zeitraum

PROCESS P1 MODEL 0.001 5      # Einschalten bzw. Ausschalten der
                              # Pulsaufzeichnung für einen spezifischen
                              # Prozess für den angegebenen Zeitraum

END_MARKER                    # Abschluss der Datei
```

10.5.4. Hostdatei

In der Hostdatei werden Partitonsnamen und damit den gleichnamigen Partitionsdateien eindeutige Nummern zugeordnet und deren Abhängigkeiten zueinander bezüglich eingehender und ausgehender Verbindungen angegeben. Des Weiteren werden den Partitionen Rechnernamen zugeordnet, wobei die Zuordnung mehrerer Partitionen zu einem Rechner möglich ist.

Für die Kommunikation zwischen den Rechnern können Optionen angegeben werden, welche die Art der Nachrichtenübermittlung, das Synchronisationsschema und die Puffermethode steuern.

Bei der Übermittlung kann zwischen der Einstellung FAST oder SAVE gewählt werden, wobei mit FAST eine unsichere Oneway Übermittlung verwendet wird.

Bei dem Synchronisationsschema können PEINQUIRIES aktiviert werden, mit denen bei einer Blockade eine Anfrage bei dem Vorgänger gestellt wird.

Die Pufferung lässt sich deaktivieren, oder unter unterschiedlichen Randbedingungen aktivieren. Es kann eine Pufferung bis zum nächsten Zeitschritt (Ein Schritt in der gewählten Zeitauflösung) erfolgen, wobei dies mit oder ohne Entleerung des Puffers bei Anfragen geschehen kann und es kann eine vollständige Pufferung erfolgen, die immer bis zur nächsten Anfrage durchgeführt wird.

Beispiel einer Hostdatei für 4 Partitionen

```
#HOST INFORMATION

[PARTITIONS]            # Schlüsselwort leitet den Abschnitt
                        # mit den Partitionsbeschreibungen ein
NUMBERPARTITIONS 4      # Gesamtzahl der zu simulierenden Partitionen

# Im folgenden Abschnitt werden die Partitionen nummeriert und benannt
# i PNAME <name>  is the name of the partition i  ( 0 <= i < n )
0 PNAME part_0
1 PNAME part_1
2 PNAME part_2
3 PNAME part_3

# Im folgenden Abschnitt werden Abhängigkeiten zwischen den
# Partitionen durch die Angabe von eingehenden und ausgehenden
# Verbindungen wiedergegeben
# i IN a b c ...  a,b,c are InputPartions of i
# i OUT a b c ... a,b,c are OutputPartions of i
0 IN 3
0 OUT 2
1 OUT 3
2 IN 0
3 IN 1
3 OUT 0

# Im folgenden Abschnitt werden den Partitionsnummern Rechnernamen
# zugeordnet
# i HOST <hostname>     shows the wish to simulate partition i on
<hostname>
0 HOST zmtmbdb3
1 HOST zmtmml60
2 HOST zmtmp148
3 HOST zmtmp243

# In diesem Abschnitt können Optionen für den Datentransfer angegeben
# werden.
# Folgende Optionen werden unterstützt:
```

```
# TRANSFERMODE    {SAVE, FAST}
# PREINQUIRIES    {1, 0}
# BUFFERMODE      {NO_BUFFERING, LVT_STEP_BUFFERING,
#                  LVT_BUFFERING_WITH_FLUSH, FULL_BUFFERING}
# i OPTION <PARAMETER> <VALUE>      will send option <PARAMETER> with
<VALUE> to Partition i
0 OPTION TRANSFERMODE SAVE
1 OPTION TRANSFERMODE SAVE
2 OPTION TRANSFERMODE SAVE
3 OPTION TRANSFERMODE SAVE

[END_PARTITIONS]        # Markiert das Ende der Partitionsbeschreibung
```

10.6. Programmierung eines Neuronenmodells

10.6.1. Klassenstruktur

Für einen Überblick über die Softwarearchitektur und das Zusammenwirken der darin definierten Klassen sei auf Abschnitt 4.2 verwiesen.

10.6.2. Namenskonventionen

Bei Klassen, die sowohl vom Benutzer implementiert werden können, als auch durch Spikelab bereitgestellt werden, führen die Spikelab-Klassen grundsätzlich „Spikelab“ im Namen (Bsp. *SpikelabAxonalOutput* für einen Ausgangstyp, der von Spikelab bereitgestellt wird).

Bei ParameterManagern soll stets ein „PM_“, bzw. ein „PMG_“ für die graphische Variante, in Verbindung mit dem Namen des logischen Prozesses verwendet werden (Bsp. *PM_Spiketrain*). Für ParameterSets gilt analog „PS_“ als Prefix vor dem Namen des logischen Prozesses (Bsp. *PS_Spiketrain*).

10.6.3. ParameterManager und ParameterSet

ParameterManager und ParameterSets werden im ParameterManager Projekt hinzugefügt. Dabei ist stets die oben angegebene Namenskonvention zu beachten. Zum ParameterManager existiert auch ein Pendant für die graphische Oberfläche. Dieses Pendant dient dazu alle zur graphischen Oberfläche relevanten Aspekte zu kapseln und greift für die funktionale Implementierung auf den nicht graphischen ParameterManager zurück. Die Schnittstelle des graphischen Pendants ist weiter unten in Abschnitt 10.6.7 beschrieben.

10.6.4. Schritte zur Implementierung

Zur Implementierung eines Neuronenmodells sollte in folgenden Schritten vorgegangen werden. Dabei ist der Bezeichner *MeinModellName* als Platzhalter zu verstehen. Um ein neues Modell zu implementieren, kann man sich den im Folgenden angegebenen Beispielen oder aus bestehenden Modellen bedienen:

1. Anlegen der Dateien *MeinModellName.cpp* und *MeinModellName.h* im Verzeichnis Spikelab\Source\Simulator\Kernels
2. Anlegen der Dateien *PM_MeinModellName.cpp* und *PM_MeinModellName.h*, sowie *PS_MeinModellName.cpp* und *PS_MeinModellName.h* im Verzeichnis

Spikelab\Source\ParameterManager, wobei meist auf *PS_MeinModellName.cpp* verzichtet werden kann.

3. Implementierung des Modellverhaltens in der Methode *evaluate* (Datei: *MeinModellName.cpp*), wobei die identifizierten Parameter jeweils in *PS_MeinModellName.h* festgehalten werden sollten.
4. Implementierung der Parameterverwaltung in *PM_MeinModellName.cpp*
5. Die genannten Dateien in die entsprechenden Projekte (*dllpm* und *SimulatorKernels*) des SPIKELAB Workspaces einfügen und übersetzen.
6. Nun gegebenenfalls für die Verwendung mit der graphischen Oberfläche die Dateien *PMG_MeinModellName.cpp* und *PMG_MeinModellName.h* im Verzeichnis Spikelab\Source\Spikelab_GUI\CustomGUI anlegen, implementieren und in das Projekt CustomGUI des SPIKELAB Workspaces aufnehmen und übersetzen.

10.6.5. Logischer Prozess (MeinModellName.cpp/h)

Im Grunde beschränkt sich die Implementierung eines Modells auf die Implementierung der Klassenmethode *evaluate*. Hierbei ist es ausreichend lediglich die Methode zu implementieren, die als Parameter ein einzelnes Ereignis akzeptiert. Fehlt die Methode, welche einen Vektor solcher Ereignisse akzeptriert, so wird von der Basisklasse wiederholt die *evaluate* Methode mit einzelnen Ereignissen aufgerufen. Sollte jedoch ein Optimierungspotential in der Verarbeitung mehrerer Ereignisse gezogen werden können, so empfiehlt sich die Implementierung der Methode, um die Simulationsleistung zu verbessern.

Entscheidend ist, dass die *evaluate* Methode, basierend auf ein eingehendes Ereignis stets alle Ereignisse in der Liste zurückliefert, die als zweiter Parameter der Methode angegeben ist. Entsteht kein ausgehendes Ereignis, so ist eine leere Liste zurückzugeben. Als Ausnahme von dieser Regel kann ein Modell ein an sich selbst gerichtetes Ereignis in dieser Liste übergeben, um z. B. sicherzustellen, dass von äußeren Ereignissen unabhängige Zustandsänderungen im Modell zum richtigen Zeitpunkt durchgeführt werden können.

Für die in SPIKELAB spezifizierten Ereignistypen und für den Benutzer spezifizierbaren Ereignistypen, sind die Dateien *Event.h* (Klasse *EventTypeRegistry*) und *const.h* (*EventTypeID* Definitionen) zu konsultieren.

Codebeispiel:

```
#ifndef __IDPROCESS_H
#define __IDPROCESS_H

#include "LpKernel.h"

class IdProcess : public LogicProcessKernel {

public:

  IdProcess() : LogicProcessKernel()
  {
    name = "IdProcess";
  }

  static const char* getClassName()
  {return "IdProcess";}
```

```
  // evaluate one event
  void evaluate(const Event&, list< pair<Event, outputID> >& ) ;

  //evaluate multiple events
  void evaluate(const vector<Event*>&, list<pair<Event, outputID>>&);

  virtual void logData(const Time&);

  void reset();
};

#endif
```

Die Implementierung des IdProzesses fällt denkbar einfach aus, da dieser Prozess lediglich jedes eingehendes Ereignis weiterleitet. Für die Implementierung der *evaluate* Methode, welche mehrerer Ereignisse verarbeitet, sei auf die Originaldatei *IdProcess.cpp* verwiesen Mit der statischen Instanz eines so genannten „Proxyobjekts“, welches die Klasse des Modells als Templateparameter verwendet, wird die automatische Einbindung des Modells in das Simulationssystem sichergestellt:

```
#include "Event.h"
#include "IDProcess.h"
#include "LogicProcess.h"

//#########################################################################
//# Declare an instance of the proxy to register the                      #
//# existence of IdProcess with the LpKernelStore.                        #
//#########################################################################
#ifdef __COMPILE_DLL
static LpKernelProxy<IdProcess> gIdProcessProxy;
#endif

void IdProcess::evaluate ( const Event& e, list< pair<Event, outputID>
>& buffer)
{
    Time la;
      if(ownerLp->getNoOfInputs() > 1)
      {
            if(ownerLp->getLocalHorizont() > ownerLp->getLVT())
                  la = ownerLp->getLocalHorizont() - ownerLp->getLVT();
            else
                  la = 0;
      }
      else
                  la = e.getLookAhead();

      pair<Event, outputID> tempPair;
      tempPair.first = Event(e.getTime(), e.getType(), la);
      tempPair.second = TO_ALL_SUCCESSORS;
      buffer.push_back(tempPair);
      // if logging is on log something
      if (logging && e.getTime() >= startLog)
      {
            if(e.getTime() < endLog)
            {
                  (*logStream) << ownerLpId << " " << e.getTime();
                  (*logStream) << " #IdProcess" << endl;
            }
            else
                  loggingOff(endLog-1);
      }
}
```

```
void IdProcess::logData(const Time& t)
{
      if(t < startLog)
            return;
      if(t == startLog)
      {
            (*logStream) <<"# " << ownerLpId << " is an IdProcesss";
            (*logStream) << endl;
            return;
      }
}

void IdProcess::reset( void )
{
      return;
}
```

10.6.6. ParameterManager (PM_MeinModellName.cpp/h)

Die Schnittstellenbeschreibung und -implementierung eines ParamaterManagers wird im Folgenden am Beispiel des Modells Spiketrain vorgestellt. Die Parameter des Modells sind in PS_Spiketrain.h definiert. Im Falle des Spiketrains handelt es sich lediglich um das Interval mit dem der Spiketrain Pulse an seinem Ausgang generiert.

Da es sich um ein Generatormodell handelt, welches nur in Abhängigkeit des zu spezifizierenden Intervalls, jedoch unabhängig von äußeren Ereignissen selber Ereignisse generiert, besitzt das Modell keine Eingangstypen.

10.6.6.1. ParameterSet

```
#ifndef __PS_SPIKE_TRAIN_H
#define __PS_SPIKE_TRAIN_H

#include "ParameterSet.h"

#ifdef _PARAMETER_MANAGER_DLL
#define PM_CLASS __declspec( dllexport )
#else
#define PM_CLASS __declspec( dllimport )
#endif

//#########################################################################
//#   BEGIN OF                Interface of class PS_SpikeTrain       #
//#########################################################################

class PM_CLASS PS_SpikeTrain: public ParameterSet
{
public:
      float interval;
public:
      PS_SpikeTrain();
      ~PS_SpikeTrain();
};

#endif
```

10.6.6.2. ParameterManager

Die Schnittstelle des ParameterManagers sieht im Wesentlichen drei unterschiedliche Zugangspunkte vor, das Core-, Input- und OutputParameterSet. Die Erzeugung und das Entfernen und Verwalten dieser ParameterSets ist die zentrale Funktion des

ParameterManagers. Die Unterteilung in Core, Input- und Outputparameter ermöglicht es Parameter eines Modells getrennt nach grundsätzlichen, gemeinsam genutzten Parametern, Eingangs- und Ausgangsparametern zu verwalten. Ein einfaches Beispiel für die sinnvolle Trennung von Modellparametern ist zum Beispiel der Schwellenwert eines Neurons und dessen synaptischen Eingangsgewichte. Der Schwellenwert existiert nur einmal, wobei es eine beträchtliche Zahl unterschiedlicher Eingangsgewichte geben kann.

Die Methoden zur Erzeugung (create…) werden jeweils aufgerufen, wenn ein Modell instanziert wird (*createCoreParameterSet* – wenn z. B. ein Neuron eingefügt wird), ein neuer Eingang an einem Neuron erzeugt wird (*createInputParameterSet*) oder ein neuer Ausgang erzeugt wird (*createOutputParameterSet*). Entsprechen werden die remove… Methoden aufgerufen, wenn einer der genannten Elemente entfernt wird.

Dem Konstruktor des Beispielcodes ist zu entnehmen, wie Ausgangstypen zu erzeugen sind, dies funktioniert auf die gleiche Weise bei Eingangstypen.

Dem Code ist weiterhin zu entnehmen, dass im *PM_Spiketrain* die ParameterSets in einer STL-Map verwaltet werden, welches es erlaubt mit Hilfe des Zeigers auf den Netzwerkknoten, der den Spiketrain repräsentiert, das entsprechende ParameterSet zu finden (siehe Methode *createCoreParameterSet*).

Da dieses Modell genau einen Ausgangstypen besitzt, dem genau ein Ausgang zugeordnet ist, bleibt die Funktion *createOutputParameterSet* leer.

Mit den Methoden *writeToStream* und *readFromStream* wird der Parameter des Modells, das Intervall, in die Gewichtsdatei geschrieben bzw. aus dieser wieder ausgelesen.

Beim Einlesen und Schreiben der Daten wird dem ParameterManager jeweils der Stream, positioniert an der Stelle übergeben, an der die Parameter des Modells gespeichert respektive gelesen werden müssen.

10.6.6.2.1. Schnittstelle eines ParameterManagers am (Spiketrain.h):

```
#include "ParameterManager.h"
#include "PS_SpikeTrain.h"

class PM_CLASS PM_SpikeTrain : public ParameterManager
{
friend class PMG_SpikeTrain;

private:
      float defaultInterval;

public:
      PM_SpikeTrain();
      virtual ~PM_SpikeTrain();
      void writeToStream(ParameterSet* /*paramSet*/, ostream& /*os*/);
      ParameterSet* readFromStream(istream& /*is*/);

// CREATION OF ENTRIES:
      virtual void createCoreParameterSet(const Node* /*Node*/);
      virtual void createInputParameterSet(Input* /*InputID*/);
      virtual void createOutputParameterSet(Output* /*OutputID*/);

// REMOVING OF ENTRIES:

      // REMOVE CORE PARAMETER SET
      virtual void removeCoreParameterSet(const Node* /*NodeID*/);
```

```
      // REMOVE INPUT PARAMETER SET
      virtual void removeInputParameterSet(const Input* /*InputID*/);
      // REMOVE OUTPUT PARAMETER SET
      virtual void removeOutputParameterSet(const Output*
/*OutputID*/);
      void fillParameterSetDefault(PS_SpikeTrain* /*pPS*/);
      const InputTypeList* getInputTypes();
      const OutputTypeList* getOutputTypes();
// MISC:
      float getDefaultInterval(){return defaultInterval;}
      void setDefaultInterval(const float& di){defaultInterval = di;}
};

#endif // !defined__PM_SPIKE_TRAIN_H
```

10.6.6.2.2. Implementierung eines ParameterManagers am (Spiketrain.cpp):

Ebenso wie für die Modelle muss auch für die automatische Erkennung des ParameterManagers in SPIKELAB ist die Instantzierung eines statischen Proxyobjekts mit dem ParameterManager als Templateparameter notwendig. Dieser Proxy erlaubt SPIKELAB die Instanzierung des ParameterManagers, ohne das SPIKELAB selbst neu übersetzt werden muss.

```
#ifdef __COMPILE_DLL
static ParameterManagerProxy<PM_SpikeTrain> gPM_SpikeTrainProxy;

#endifPM_SpikeTrain::PM_SpikeTrain():ParameterManager()
{
      Generator = true;
      name = "PM_SpikeTrain";
      defaultInterval = 0.25f;

      inTypes = new InputTypeList();
      outTypes = new OutputTypeList();
      // INPUT_TYPES
            // no input types
      // OUTPUT_TYPES
      SpikelabAxonalOutput* output = new SpikelabAxonalOutput;
      const type_info& ti = typeid(*output);
      string typeName = ti.name();
      OutputType* outT = new OutputType(typeName, output);
      outTypes->push_back(outT);
      delete output;
}

PM_SpikeTrain::~PM_SpikeTrain(){}

void PM_SpikeTrain::fillParameterSetDefault(PS_SpikeTrain* pPS)
{
      pPS->interval = 0.25f;
}

void PM_SpikeTrain::createCoreParameterSet(const Node* nd)
{
      PS_SpikeTrain* pPS = new PS_SpikeTrain();
      fillParameterSetDefault(pPS);
      coreParamSets.insert(map<Node*,
      ParameterSet*>::value_type((Node*)nd, pPS));
}

void PM_SpikeTrain::createInputParameterSet(Input* i)
{
}
```

```
void PM_SpikeTrain::createOutputParameterSet(Output* output)
{
}
void PM_SpikeTrain::writeToStream(ParameterSet* paramSet, ostream& os)
{
      // Just put the known parameter into the stream
      os << "$INTERVAL$ " << ((PS_SpikeTrain*)paramSet)->interval;
      os << endl;
}

ParameterSet* PM_SpikeTrain::readFromStream(istream& is)
{
      PS_SpikeTrain* paramSet = new PS_SpikeTrain();
      paramSet->interval = (float)0.001;
      string strBuf;
      streampos pos = is.tellg();
      is >> strBuf;

      while(is.good() && strBuf != "$PROCESS$" && (strBuf !=
                  "$END_MARKER$" && strBuf != "END_MARKER" ))
      {
            if(strBuf == "$INTERVAL$")
                  is >> paramSet->interval;
            else
            {
                  _ERROR_STREAM << "ERROR: UNVALID *DATA* for;
                  _ERROR_STREAM << "SPIKE_TRAIN ";
                  _ERROR_STREAM << endl;
                  _ERROR_STREAM << "UNVALID TOKEN FOUND: " << strBuf;
                  _ERROR_STREAM << endl;
                  return paramSet;
            }
            pos = is.tellg();
            is >> strBuf;
      }
      is.seekg(pos);
      return paramSet;
}

void PM_SpikeTrain::removeCoreParameterSet(const Node* node)
{
}

void PM_SpikeTrain::removeInputParameterSet(const Input* input)
{
}

void PM_SpikeTrain::removeOutputParameterSet(const Output* output)
{
}

const InputTypeList* PM_SpikeTrain::getInputTypes()
{
      return inTypes;
}

const OutputTypeList* PM_SpikeTrain::getOutputTypes()
{
      return outTypes;
}
```

10.6.7. GUI-Klassen

Das graphische Pendant zur Klasse *ParameterManager* ist die Klasse *ParameterManagerGUI* (*PMG_MeinModellName.cpp/h*). Sie dient dazu über die graphische Oberfläche auf die Parameter zuzugreifen, die vom ParameterManager verwaltet werden. Zum Entwurf eines entsprechenden Dialogs sollte der Resourcen-Editor des Microsoft Developer-Studios verwendet werden. Es bietet sich an, die Klasse *PMG_MeinModellName* zur *friend*-Klasse in der Klasse *PM_MeinModellName* zu deklarieren, um die Definition vieler Methoden zu vermeiden, die nur zur Übergabe von Daten zwischen diesen beiden Klassen dienen.

Die Schnittstelle des graphischen ParameterManagers enthält drei Methoden, welche Dialoge für die Bearbeitung von Parametern aufrufen, die den Kern, die Eingänge bzw. Ausgänge betreffen.

Die Benennung der Dialoge muss der Namenskonvention *DlgMeinModellName* bzw. *DlgInMeinModellName* und *DlgOutMeinModellName* folgen. Die Implementierung der Dialoge für Eingangsparameter und Ausgangsparameter kann, wie im Beispiel je nach Typ des Modells auch entfallen. Es reicht eine leere Methode zu implementieren, jedoch bietet es sich an zumindest für die Testphase der Implementierung Ersatzausgaben vorzusehen, wie sie auch im Beispiel erfolgen.

Auf den Abdruck der Dialogimplementierung wurde verzichtet, da diese einigen, zur MFC spezifischen Code enthält. Hier sei direkt auf die Quellen verwiesen.

10.6.7.1. Schnittstelle eines graphischen ParameterManagers (PMG_Spiketrain.h)

```
#include "ParameterManagerGUI.h"

class MODEL_CLASS PMG_SpikeTrain : public ParameterManagerGUI
{
      float defaultInterval;
public:

  PMG_SpikeTrain();
  virtual ~PMG_SpikeTrain();

  void runDialog(NodeList* nl);
  void runInDialog(InputList* nl);
  void runOutDialog(OutputList* nl);
};
```

10.6.7.2. Implementierung eines graphischen ParameterManagers (PMG_Spiketrain.cpp)

Wie für den ParameterManager selber ist es auch für das graphische Pendant notwendig ein eine statische Proxyinstanz anzugeben.

```
//#########################################################################
//# Declare an instance of the proxy to register the                      #
//# existence of PMG_SpikeTrain with the ParameterManagerGUIStore.        #
//#########################################################################
static ParameterManagerGUIProxy<PMG_SpikeTrain> gPMG_SpikeTrainProxy;

PMG_SpikeTrain::PMG_SpikeTrain():ParameterManagerGUI()
{
  name = "PMG_SpikeTrain";
  dialog = new CDlgSpikeTrain();
```

```
}

PMG_SpikeTrain::~PMG_SpikeTrain()
{
  if(dialog)
  {
    delete dialog;
    dialog = NULL;
  }
}

void PMG_SpikeTrain::runDialog(NodeList* nl)
{
  float currInterval = ((PM_SpikeTrain*)parameterManager)-
>getDefaultInterval();
  ((CDlgSpikeTrain*)dialog)->m_all_equal = false;
  ((CDlgSpikeTrain*)dialog)->m_globalSettings = true;
  ((CDlgSpikeTrain*)dialog)->m_Interval = currInterval;

  // Showing up the dialog:
  dialog->DoModal();

  if(((CDlgSpikeTrain*)dialog)->m_Interval_edited)
  {
    ((PM_SpikeTrain*)parameterManager)-
>setDefaultInterval(((CDlgSpikeTrain*)dialog)->m_Interval);
  }
}

void runInDialog(InputList* il)
{
  // This model has no inputs
  // Use message box for checks, otherwise an empty method is
  // sufficient
  String message = String("PMG_SpikeTrain::runInDialog: ")
      + String("This model has no inputs!")
  AfxMessageBox(message.chars(), MB_OK);
  return;
}
void runOutDialog(OutputList* ol)
{
  // This model has an output, but there are no parameters to set
  // Use message box for checks, otherwise an empty method is
  // sufficient
  String message = String("PMG_SpikeTrain::runOutDialog: ")
    + String("This model has no settable parameters for its output!")
  AfxMessageBox(message.chars(), MB_OK);
  return;
}
```

10.7. Programmierung eines Netzwerkgenerators

10.7.1. Namenskonventionen

Bei Netzwerkgeneratoren soll stets ein „NG_“ in Verbindung mit dem Namen des Netzwerkgenerators verwendet werden (Bsp. *NG_LayerNetgenerator*). Für die Dialoge gilt dem entsprechend „Dlg“ als Prefix des Namens (*DlgLayerNetgenerator*).

Netzwerkgeneratoren und deren Dialoge werden als Dateien in dem Verzeichnis Spieklab\Source\SpikelabGUI\netgen abgelegt und dem *netgendll* Projekt im Spikelab Workspace hinzugefügt. Dabei ist stets die oben angegebene Namenskonvention zu beachten.

10.7.2. Schnittstelle eines Netzwerkgenerators

Für eine minimale Implementierung eines Netzwerkgenerators muss lediglich die Methode *rundDialog* implementiert werden. In dieser Methode ist der selbst entworfene Dialog aufzurufen, mit dem die Parameter abgefragt werden, die für die Netzwerkgenerierung benötigt werden.

In der Regel ist es sinnvoll, die Generierung in *private* deklarierte Methoden zu unterteilen, was wiederum sehr von der zu generierenden Netzwerkstruktur abhängt. Beim *LayerNetGenerator* findet sich vor allem eine Unterteilung in den Aufbau (*build_topology*) und die Verknüpfung (*connect*) der Netzwerkstruktur. Die Methoden *setCurrentDialogData* und *updateFromDialogData* sind private Helfermethoden der Methode *runDialog*.

```
class NG_LayerNetGenerator : public NetworkGenerator
{
private:
      void setCurrentDialogData(const Node* /*rootNode*/);
      void updateFromDialogData();
      void genNetwork();
      void Init();
      void resize(unsigned int m_n_layer,
                  float _lx, float _ly, float _lz);
      void build_topology(unsigned int _nx, unsigned int _ny);
      void connect();

public:
      NG_LayerNetGenerator();
      virtual ~NG_LayerNetGenerator();
      void runDialog(const Node* /*rootNode*/);
      void setBatchParam(NetGenBatchParam* pBp);
      void batchGen(const Node* /*rootNode*/);
};
```

10.7.3. Implementierung eines Netzwerkgenerators

Die Implementierung des *LayerNetGenerators* wird hier nur ausschnittsweise wiedergegeben, da die gesamte Implementierung mehrere Seiten füllen würde. Zu sehen ist der Konstruktur, indem der Dialog für den Generator instanziert wird und die Methode *runDialog*, welche sich der privaten Helfermethoden *setCurrentDialogData* und *updateFromDialogData* bedient.

Wie beim ParameterManager muss ein Proxyinstanz erzeugt werden, damit der Netwerkgenerator beim Programmstart automatisch in dass Menü der Netzwerkgeneratoren aufgenommen wird.

```
//#####################################################################
//# Declare an instance of the proxy to register the                  #
//# existence of NG_LayerNetGenerator with the NetworkGeneratorStore.
//#####################################################################
#include "NetworkGeneratorProxy.h"
static                      NetworkGeneratorProxy<NG_LayerNetGenerator>
gLayerNetGeneratorProxy;

NG_LayerNetGenerator::NG_LayerNetGenerator():NetworkGenerator()
{
      name="LayerNetGenerator"; // The Name for the Dynanic-Menue
      dialog = new CDlgLayerNetGenerator();
      Init();
}

void NG_LayerNetGenerator::setCurrentDialogData(const Node* rootNode)
```

```
{
	root=rootNode;
	dialog->m_Layer_lx=m_lx;
	dialog->m_Layer_ly=m_ly;
	dialog->m_Layer_lz=m_lz;
	dialog->m_Layer_nx=m_nx;
	dialog->m_Layer_ny=m_ny;
	dialog->m_Layer_n=m_n_layer;
	dialog->m_Layer_name=baseName;
	dialog->m_Layer_Connect=Connect;
	dialog->m_bAddSpikeTrain = bAddSpikeTrain;
	dialog->m_fSpikeTrainInterval = fSpikeTrainInterval;
	dialog->m_bRandomizeDelays = bRandomizeDelays;
	dialog->m_bRandomizeWeights = bRandomizeWeights;
	dialog->m_fDelayOffset = fDelayOffset;
	dialog->m_fDelayRange = fDelayRange;
	dialog->m_fWeightNodeSum = fWeightNodeSum;
	dialog->m_fWeightRange = fWeightRange;
	dialog->m_Layer_Layout=0;
	dialog->m_ModelID   =   m_ModelID   =   gNetgenDLL_Imports.pPMGS-
>getModelId("StdModIAFf");
	//OnSelchangeListModel();

}

void NG_LayerNetGenerator::updateFromDialogData()
{
	baseName=dialog->m_Layer_name;
	m_lz=dialog->m_Layer_lz;
	m_lx=dialog->m_Layer_lx;
	m_ly=dialog->m_Layer_ly;
	m_nx=dialog->m_Layer_nx;
	m_ny=dialog->m_Layer_ny;
	m_n_layer=dialog->m_Layer_n;
	m_ModelID=dialog->m_ModelID;
	Connect=dialog->m_Layer_Connect;
	bAddSpikeTrain = dialog->m_bAddSpikeTrain > 0 ? true : false;
	fSpikeTrainInterval = dialog->m_fSpikeTrainInterval;
	bRandomizeDelays  =  dialog->m_bRandomizeDelays  >  0  ?  true  :
false;
	bRandomizeWeights  =  dialog->m_bRandomizeWeights  >  0  ?  true  :
false;
	fDelayOffset = dialog->m_fDelayOffset;
	fDelayRange = dialog->m_fDelayRange;
	fWeightNodeSum = dialog->m_fWeightNodeSum;
	fWeightRange = dialog->m_fWeightRange;
	Layout= dialog->m_Layer_Layout;
  }

void NG_LayerNetGenerator::runDialog(const Node* rootNode)
{
	setCurrentDialogData(rootNode);
	dialog->DoModal();
	updateFromDialogData();
	if(!dialog->m_Canceled)
	{
		SetCursor(AfxGetApp()->LoadCursor(IDC_BUSY_CURSOR));
		genNetwork();
		SetCursor(AfxGetApp()->LoadCursor(IDC_ARROW));
	}
}
```

Kapitel 11 - Abbildungsverzeichnis

Abbildung 1: Aufbau des Gehirns (aus [Spektrum1988]) 12
Abbildung 2: Funktionale Bereiche der Großhirnrinde (aus [Spektrum1988]). 13
Abbildung 3: Homunkulus - Karte der Körperteile auf dem somato-sensorischen und dem motorischen und Rindenfeld (aus[Spektrum1988]) 14
Abbildung 4: Schichten des cerebralen Cortex (aus [Thompson1994]) 15
Abbildung 5: Neuronenanordnung und Vernetzungsarten (aus [Schüz1995]) 15
Abbildung 6: Neuronenarten (nach Ramón y Cajal, 1933 - aus [KanSchJes1995]) 16
Abbildung 7: Aktionspotential (aus [Schmidt1987]) 18
Abbildung 8: Formen des Aktionspotentials (aus [Schmidt1987]) 18
Abbildung 9: Messung des Aktionspotentials einer Nervenzelle aus einem Fliegengehirn: Oben der abgeleitete Spannungsverlauf; Unten: Der Bandpassgefilterte Verlauf, wobei auf der rechten Seite fünf Aktionspotentiale überlagert auf einer gestreckten Zeitachse dargestellt sind. (aus [RieWarRuy1997]) 19
Abbildung 10: Verlauf des exzitatorischen und inhibitorischen Potentials 20
Abbildung 11: Ersatzschaltbild des Hodgkin-Huxley-Modells 25
Abbildung 12: Prinzipielles Vorgehen bei der Erstellung eines Abschnitts-Modells. Aus neurophysiologischen Messungen werden Parameter für ein Kabelmodell gewonnen (A) und das Neuron wird mit seinen Fortsätzen durch kurze Kabelabschnitte modelliert (B). Diese Kabelabschnitte werden anschließend durch äquivalente elektronische Schaltkreise ersetzt (C). (aus [BowBee1998]) 26
Abbildung 13: Schaltbild und Feuerrate eines Integrate-and-Fire-Neuron 27
Abbildung 14: Typische Verläufe (a) der Eingangsfunktion (postsynaptisches Potential) und (b) der Ausgangsfunktion (Nachpotential) in einem Spike-Response-Modells. 28
Abbildung 15: Aufbau des Eckhorn-Neuronenmodells 31
Abbildung 16: Die postsynaptische Eingangsfunktion des Eckhorn-Modells weist einen instantanen Anstieg auf und setzt nach der Summe Δ_{ges} aus axonaler und synaptischer Verzögerung ein. 32
Abbildung 17: Raum-Zeit Diagramm für eine (a) zeitgetriebene und (b) ereignisgetriebene zeitdiskrete Simulation (nach [Fujimoto2000]) 38
Abbildung 18: Senderorientierte und empfängerorientierte Ereignisverteilung 40
Abbildung 19: Vergleich der STL Priority und der Calendar Queue für Füllstände bis 200.000 Elemente. 44
Abbildung 20: Vergleich der STL Priority und der Calendar Queue für Füllstände bis 10.000 Elemente. 44
Abbildung 21: Calendar Queue Struktur mit 4 Monaten und einer Monatsbreite von 25 Tagen. Ein Kalenderjahr dauert in diesem Kalender also 100 Tage. 46
Abbildung 22: Partitionierung eines Netzwerks auf mehrere Rechner 51
Abbildung 23: Verteilung der Ereignisse durch den Sender oder den Empfänger 52
Abbildung 24: Quellen numerischer und verfahrensbedingter Abweichungen 54
Abbildung 25: Verschiebung des Feuerzeitpunkts durch ein erneut eintreffendes Ereignis. 60
Abbildung 26: Direkte und indirekte Verbindungen von Prozessen 61

Abbildung 27: Sicheres Fenster für die Verarbeitung von Ereignissen in einer konservativ synchronisierten Simulation 67
Abbildung 28: Bestimmung des ausgangsseitigen look aheads mit Hilfe von Verzögerungen 68
Abbildung 29: Transferzeiten und –raten für PC-PC (Windows) Kommunikation. 77
Abbildung 30: Transferzeiten und -raten für SUN - SUN (Solaris) Kommunikation 78
Abbildung 31: Transferzeiten und -raten für SUN - PC (Solaris - Windows) Kommunikation 79
Abbildung 32: Einfluss des Packing bei PVM 80
Abbildung 33: Eingabefenster von SPIKELAB. Links sind die beiden Baumstrukturen zu erkennen, welche beide die Hierarchie der aktuell betrachteten Topologie wiedergeben. Rechts ist die Liste der Verbindungen zu sehen, die durch die Selektion der Startneuronen im linken Baum und den Zielneuronen im rechten Baum ausgewählt wurden. (Visualisiert ist ein 3-schichtiges Netzwerk mit je 25 Neuronen pro Schicht, wobei jedes Neuron mit allen Neuronen der benachbarten Schicht verbunden ist.) 85
Abbildung 34: 3D-Visualisierung eines dreischichtigen Netzwerks in SPIKELAB. Oben links (a) ist die oberste Hierarchieebene in Form eines das gesamte Netzwerk umhüllenden Quaders zu sehen. Oben rechts (b) zeigt die nächste Hierarchieebene, in der die Umhüllenden der einzelnen Neuronenschichten erkennbar sind. Die Verbindungen zwischen den Schichten geben an, dass grundsätzlich Verbindungen zwischen Neuronen der Schichten existieren. Unten links (c) ist eine der drei Schichten expandiert worden. Hier sind einzelne Neuronen und deren Verbindungen zu den nächstgelegenen Kontaktpunkten der Nachbarschicht sichtbar. Unten rechts (d) sind alle Schichten expandiert und sämtliche Neuronen und deren Verbindungen dargestellt. 86
Abbildung 35: Eine Topologie bestehend aus Knoten, die Netzwerke oder Neuronen repräsentieren. Die simulierbare Repräsentation eines Knotens ist der logische Prozess. Die logischen Prozesse werden in Partitionen zusammengefasst. 89
Abbildung 36: Vererbungshierarchie der unterschiedlichen Simulatorklassen. 89
Abbildung 37: Subsimulator 91
Abbildung 38: Modellierung eines Neurons durch LogicProcessKernel und FanOutConnection 91
Abbildung 39: Vererbungshierarchie des logischen Prozesses 92
Abbildung 40: Generatorprozesse in Spikelab 95
Abbildung 41: Verlauf des postsynaptischen Potentials im Modellneuron 97
Abbildung 42: Fünfschichtiges, vollvernetztes Testnetzwerk mit SpikeTrain als speisendem Prozess 98
Abbildung 43: Vergleich der ereignisoptimierten GENESIS-Simulation und der Standardimplementierung (Aktivität: 0.25 Pulse pro Sekunde und Neuron, Auflösung r = 1E-5) 100
Abbildung 44: Abhängigkeit der Simulationszeiten von der Aktivität. Simulationen eines 5-lagigen Netzwerks mit 100 Neuronen pro Schicht und einer durch das Generatorneuron eingeprägten Aktivität von 0,25 und 4 Pulsen pro Sekunde in GENESIS und in SPIKELAB zusätzlich mit 16 Pulsen pro Sekunde. Links absolute Messwerte und rechts um den durch die Netzwerkgröße bedingten Anstieg der Aktivität bereinigte Messwerte. 101
Abbildung 45: Simulationszeiten für ein 5-lagiges Netzwerk, mit 100 Neuronen pro Schicht und unterschiedlichen eingeprägten Aktivitäten – links zwischen 0,25 und 4 bzw. 16 Pulsen pro Sekunde für die SPIKELAB-Simulation und rechts zwischen 0,25

und 4 Pulsen pro Sekunde für die GENESIS Simulation (Standardverfahren und ereignisoptimiertes Verfahren) . *101*
Abbildung 46: Veränderung des Summenpotentials durch ungenaue Auflösung der Pulszeitpunkte *102*
Abbildung 47: Effektive Aktivität in Abhängigkeit der zeitlichen Auflösung, bei einer eingeprägten Aktivität von 4 Pulsen pro Sekunde während einer GENESIS Simulation *103*
Abbildung 48: Abhängigkeit der Simulationsdauer von der zeitlichen Auflösung für ein 5-schichtiges Netzwerk mit 100 Neuronen pro Schicht (insgesamt 501 Neuronen) und einer eingeprägten Aktivität von 4 Pulsen pro Sekunde (doppelt logarithmisch skaliert) *104*
Abbildung 49: a) voll vernetztes Netzwerk b) Speicherbelegung durch Ereignisse mit $\Delta_1<\Delta_2<\Delta_3<\Delta_n$ *105*
Abbildung 50: Verarbeitete Pulse pro Sekunde für ein voll vernetztes Netzwerk und ein geschichtetes Netzwerk. Links über die Zahl der Neuronen im Netzwerk und rechts über die Zahl der Eingänge im Netzwerk aufgetragen. *106*
Abbildung 51: Maximale Blockgröße in Abhängigkeit der Netzgröße. *107*
Abbildung 52: Einfluss der cache-Größe auf die Simulationsleistung (voll vernetztes Netzwerk) *107*
Abbildung 53: Einfluss der cache-Größe auf die Simulationsleistung (geschichtetes Netzwerk) *108*
Abbildung 54: Simulationsleistung mit priority queue (pq) und calendar queue (cq), basierend auf dem STL vector (vec) und der STL list (list) - vollvernetztes Netzwerk auf Rechner B, Aktivität 16 Pulse pro Sekunde *109*
Abbildung 55: Prozentualer Mehraufwand der ereignisgetriebenen Simulation mit und ohne gebündelte Verarbeitung für ein Netzwerk mit 11 Schichten und einer Aktivität von 16 Pulsen pro Sekunde auf Rechner B. *110*
Abbildung 56: Simulationsleistung mit und ohne gebündelte Verarbeitung für ein Netzwerk mit 11 Schichten und einer Aktivität von 16 bzw. 0,25 Pulsen pro Sekunde auf Rechner B, priority queue. *110*
Abbildung 57: Simulationsleistung mit und ohne gebündelte Verarbeitung für ein Netzwerk mit 11 Schichten und einer Aktivität von 16 bzw. 0,25 Pulsen pro Sekunde auf Rechner B, calendar queue. *111*
Abbildung 58: Bereiche konstanter Simulationsleistung: Links geschichtetes Netzwerk, rechts voll vernetztes Netzwerk auf Rechner B, jeweils mit und ohne gebündelte Verarbeitung, priority queue. *112*
Abbildung 59: Bereiche konstanter Simulationsleistung: Links geschichtetes Netzwerk, rechts voll vernetztes Netzwerk auf Rechner C, jeweils mit und ohne gebündelte Verarbeitung, calendar queue. *112*
Abbildung 60: Testnetz zur Bestimmung der Übertragungsleistung in der verteilten Simulation. *114*
Abbildung 61: Beschleunigung der Simulation eines geschichteten Netzwerks durch Verteilung der Simulation auf zwei und drei Rechnersysteme, links bezogen auf die schnellste Einzelplatzsimulation und rechts bezogen auf den Mittelwert der Einzelplatzsimulationen. *116*
Abbildung 62: Beschleunigung der Simulation eines geschichteten Netzwerks durch Verteilung der Simulation auf vier und fünf Rechnersysteme, links bezogen auf die schnellste Einzelplatzsimulation und rechts bezogen auf den Mittelwert der Einzelplatzsimulationen. *117*

Abbildung 63: Beschleunigung der Simulation eines voll vernetzten Netzwerks durch Verteilung der Simulation auf zwei und drei Rechnersysteme, links bezogen auf die schnellste Einzelplatzsimulation und rechts bezogen auf den Mittelwert der Einzelplatzsimulationen. 118
Abbildung 64: Beschleunigung der Simulation eines voll vernetzten Netzwerks durch Verteilung der Simulation auf vier und fünf Rechnersysteme, rechts bezogen auf die schnellste Einzelplatzsimulation und links bezogen auf den Mittelwert der Einzelplatzsimulationen. 118
Abbildung 65: Simulation in Echtzeit mit Hilfe der verteilten Simulation. (551 Neuronen Netzwerk – Echtzeitbedingungen bei einer Verteilung auf vier Rechner) Simulierte Zeit: 10 Sekunden. 119
Abbildung 66: Simulation eines voll vernetzten Netzwerks mit 600 Neuronen und eines geschichteten Netzwerks mit 3301 Neuronen auf ein bis fünf Rechner. 120
Abbildung 67: Blockschaltbild des Spike128k Systems aus [Wolff2001] 123
Abbildung 68: ParSPIKE Systemkonzept (übernommen aus [Wolff2001]). 125
Abbildung 69: System mit NESPINN Chip (übernommen aus [JahRotKla1996]) 125
Abbildung 70: Blockschaltbilder des MASPINN-Systems und des darin enthaltenen NeuroPipe-Chips (aus [GraSchWol2002]) 126
Abbildung 71: Blockschaltbild der RACER – Beschleunigerkarte. 129
Abbildung 72: RACER PCI – Erweiterungskarte, bestückt mit einem Modul auf dem ersten Modulsteckplatz. 130
Abbildung 73: Messungen mit idealisierter Hardware-Implementierung des Calendarqueuealgorithmus aufgetragen über die Anzahl der in der Queue enthaltenen Ereignisse. 137
Abbildung 74: Simulationsleistung der Softwareimplementierung und der Hardware-Implementierung, mit und ohne gesammelte Verarbeitung der Ereignisse (aus [Bubolz2002]). 138
Abbildung 75: Bandbreite für Datentransfers vom Host zur RACER-Hardware (DLL und PT) und von der RACER-Hardware in Speicher des Hosts (MT). (DLL, PT und MT – siehe Text) 139
Abbildung 76: Lage und Belegung der Konfigurationsschalter (DIP-Switches) auf der RACER-Hardware 147
Abbildung 77: Startdialog von Spikelab 157
Abbildung 78: Dokumentenfenster einer leeren Topologie (hier: Topology10) 158
Abbildung 79: Kontextmenü zum Hinzufügen, Entfernen und Modifizieren von Neuronen und Ordnern 159
Abbildung 80: Dialog zum Hinzufügen von Neuronen. 159
Abbildung 81: Eingangs- und Ausgangstypen von Neuronen 159
Abbildung 82: Visualisierung konkreter Eingänge und der Verknüpfungen zwischen Eingängen und Ausgängen 160
Abbildung 83: Der Aufruf eines Netzwerkgenerators erfolgt über das Menü „Tools“→“Network Generators“ 161
Abbildung 84: Dialog des Netzwerkgenerators für geschichtete Netzwerke 161
Abbildung 85: Aufruf des Simulationsdialogs über das Menü Simulation→Status 162
Abbildung 86: Simulationsdialog 163

Kapitel 12 - Tabellenverzeichnis

Tabelle 1: Datenfelder eines Ereignisses 41
Tabelle 2: Simulationszeiten für verschiedene Festkommadarstellungen und Zeitauflösungen 56
Tabelle 3: Simulationszeiten für verschiedene, normalisierte Fließkommadarstellungen und Zeitauflösungen 56
Tabelle 4: Latenz verschiedener Übertragungen in Millisekunden (Paketgröße jeweils 4 Byte) 76
Tabelle 5: Eckdaten der zur Simulation verwendeten Rechnersysteme 107
Tabelle 6: Leistungsmessungen aus [Wolff2001] mit einem Netzwerk, welches im Mittel 28.269 Eingangspulse pro ms generiert – hier ergänzt um die Zahl der verarbeiteten Pulse pro Sekunde. 113
Tabelle 7: Relative Erhöhung der Transferzeit innerhalb des Simulators, verglichen zur Messung des Datentransfers ohne Simulator. (B. steht für blockierendes Senden/Empfangen) 115
Tabelle 8: Kenndaten der Rechner, auf welche die Simulation verteilt wurde. 116
Tabelle 9: Vergleich der Simulationsgeschwindigkeiten (übernommen aus [Wolff2001]). 127
Tabelle 10: Liste der RACER-Befehle 134

Kapitel 13 - Glossar

Alles-oder-Nichts-Prinzip	Aktionspotentiale entstehen bei Neuronen nach dem Alles-oder-Nichts-Prinzip. Das heißt, dass ein Neuron entweder ein Aktionspotential mit voller Amplitude aussendet oder inaktiv bleibt. Mittlere Aktivierungen wurden bisher nicht beobachtet.
Axon	Das Axon ist der Ausgangsseitige Fortsatz eines Neurons, welcher dazu dient ein vom Neuron erzeugtes Aktionspotential an seine Nachfolger zu verteilen. Axone können sich verzweigen und bilden an Ihren Enden Synapsen aus, die entweder auf den Dendriten oder direkt auf dem Soma des nachfolgenden Neurons aufsetzten.
FPGA	Field Programmable Gate Array
Integrate-and-Fire-Modell	Neuronenmodell, welches mit einem verlustbehafteten Integrator → Leaky integrator die Eingangssignale aufintegriert und einen Ausgangspuls erzeugt, sobald eine (dynamische) Schwelle für den Integrationswert erreicht wird. Die verallgemeinerte Form dieses Modell ist das → *Spike-Response-Modell*.
Leaky integrator	Verlustbehaftete Integration. Der Ausgabewert dieses Integrators nimmt kontinuierlich ab, sofern nicht ein minimaler Eingangswert vorliegt.
Myelinhülle	Ummantelung eines Axons, welche die Geschwindigkeit, mit der das Aktionspotential fortgeleitet wird deutlich erhöht. Alle Axone in der weißen Substanz des cerebralen Cortex sind myelinisiert.
Neurotransmitter	Substanz, die in chemischen Synapsen zur Übertragung des präsynaptischen Aktionspotentials dient. Ein präsynaptisches Aktionspotential löst so genannte Vesikel aus, die den Neurotransmitter enthalten. Der Transmitter wird also in Quanten ausgeschüttet und trifft auf Rezeptoren, die in der Zellmembran eingebaut sind. Je nach Rezeptor wird dadurch in der Zelle ein erregendes oder hemmendes Potential ausgelöst. Der dominierende Neurotransmitter im cerebralen Cortex ist das Acetylcholin (ACh)
Partition	Teilabschnitt eines pulsverarbeitenden neuronalen Netzwerks.
Partitionierung	Aufteilung eines Netzes in Teilnetze. Die automatische Partitionierung ermöglicht die Zer- und Verteilung von

	sehr großen und nicht regelmäßig vernetzten neuronalen Netzen im Simulator, wobei eine weitgehend optimale Verteilung der Rechenlast erreicht wird.
PLD	Programmable Logical Device
PVNN	Abkürzung für pulsverarbeitende neuronale Netze. Die Bezeichnung für diese Art von neuronalen Netzen ist nach wie vor nicht eindeutig. Im englischen finden sich Bezeichnungen wie "Spiking Neural Networks", "Pulsed Neural Networks", "Pulsed Coupled Neural Networks (PCNN)".
Refraktärphase	Phase in der ein Neuron auch unter erneuter Erregung entweder gar nicht feuert (absolute Refraktärphase) oder nur bei sehr intensiven Reizen feuert (relative Refraktärphase). Die Phase hat meist ein zeitabhängigen Verlauf.
Spike	Idealisierte Form des Impulses (Aktionspotentials), welcher von einem Neuron ausgesendet wird, wenn das innere Potential einen Schwellenwert überschreitet. Der Impuls ist insoweit idealisiert, als er nur den Zeitpunkt beschreibt, zu dem das Aktionspotential auftritt, jedoch nicht dessen Form. Die Breite des Spikes ist letzten Endes durch die Zeitauflösung des Systems gegeben. Für ein diskretes System mit festgelegter Zeitauflösung besitzt der Spike demnach keine Ausdehnung.
Spike-Response-Modell	Modellierung eines Neurons als homogene Funktionseinheit, welche Pulse (*Spikes*) generiert, sofern eine ausreichende Erregung des Neurons vorliegt. In der Regel wird die Erregung aus der Summe der postsynaptischen Potentiale errechnet und durch Vergleich mit einem statischen oder dynamischen Schwellenwert entschieden, ob das Neuron feuert, also einen Puls aussendet. Das Spike-Response-Modell (SRM) ist im Grunde ein Integrate-and-Fire-Modell. Da es jedoch recht unterschiedliche Modelle gibt, die für sich die Bezeichnung Integrate-and-Fire-Neuron beanspruchen, wurde von Wulfram Gerstner diese Bezeichnung eingeführt. Die Menge der Spike-Response-Modelle schließt alle Varianten ein, die unter dem Begriff Integrate and Fire Modell bekannt sind. (→ Abschnitt 2.2.4)
VHDL	**V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage. VHDL dient zur Beschreibung digitaler Schaltkreise und kann mit entsprechenden Simulatoren ausgeführt werden. Aus einer Untermenge von VHDL kann mit Hilfe von Synthesewerkzeugen ein in Hardware implementierbarer Schaltkreis generiert werden.
Zeitscheibenverfahren	Beim Zeitscheibenverfahren werden alle Knoten eines Netzes in jedem Zeitschritt der Simulation aktualisiert. Die Länge eines Zeitschritts ist durch die zeitliche

Auflösung bestimmt, d.h. durch die Genauigkeit der Simulation. Dies führt dazu, dass ein linearer Anstieg der Simulationszeit zu verzeichnen ist, wenn die Zeitauflösung der Simulation erhöht wird. Diese Eigenart beschränkt diese Simulation auf relativ kleine Netze oder erfordert eine gröbere Zeitauflösung für große Netze, soll die Simulationszeit in einem erträglichen Rahmen gehalten werden.

Kapitel 14 - Eidesstattliche Erklärung

An Eides statt versichere ich, dass ich die Arbeit

„Simulation pulsverarbeitender neuronaler Netze“

unter der Leitung von Herrn Professor Dr. Anlauf und Herrn Professor Dr. Strelen als Korreferenten selbst und ohne jede unerlaubte Hilfe angefertigt habe, dass diese oder eine ähnliche Arbeit noch keiner anderen Stelle zur Prüfung vorgelegen hat und dass sie an den nachstehend aufgeführten Stellen auszugsweise veröffentlicht worden ist.

Grassmann Cyprian, Anlauf Joachim K. (1998). *Distributed, Event Driven Simulation of Spiking Neural Networks*. Neural Computation 98: 100-105.

Grassmann Cyprian, Anlauf Joachim K. (1999). *Fast Digital Simulation Of Spiking Neural Networks And Neuromorphic Integration With Spikelab*. International Journal of Neural Systems, Vol. 9, No. 5.

Grassmann Cyprian, Schönauer Tim, Wolff Carsten (2002). *PCNN Neurocomputers – Event Driven and Parallel Architectures*. Proceedings of the European Symposium on Artificial Neural Networks, Brügge, Belgien.

_____

(Cyprian Graßmann)

Kapitel 15 - Zusammenfassung

In dieser Arbeit werden die Eigenschaften von pulsverarbeitenden Neuronalen Netzen (PVNN) untersucht und eine daraufhin optimierte Lösung in Form des Simulationssystems SPIKELAB vorgestellt. SPIKELAB implementiert eine ereignisgetriebene und verteilte Simulation, welche die Eigenschaften pulsverarbeitender neuronaler Netze, wie z. B. die geringe Aktivität dieser Netzwerke, deren spärliche Vernetzung und die Verzögerung zwischen Neuronen, explizit behandelt und ein dadurch gegebenes Optimierungspotential ausschöpft. Zudem werden die Pulse im Netzwerk alleine durch den Zeitpunkt ihres Auftretens modelliert, wodurch trotz der in SPIKELAB eingesetzten Optimierungen, ohne Eingriff in das Simulationsverfahren, nahezu beliebige Berechnungsmodelle für Neuronen implementiert werden können.

Ein quantitativer Vergleich mit einer Zeitscheibensimulation für pulsverarbeitende neuronale Netzwerke zeigt, dass durch eine ereignisgetriebene Simulation deutlich kürzere Simulationszeiten für die Simulation erreicht werden. Außerdem wird durch quantitative Messungen nachgewiesen, dass feinere Zeitauflösungen nahezu keinen Einfluss auf die Simulationsleistung der ereignisgetriebenen Simulation haben, während feinere Zeitauflösungen bei der Zeitscheibensimulation mit untragbar hohen Laufzeiten erkauft werden müssen.

Zur Beschleunigung und zur Simulation großer Netzwerke wird in SPIKELAB die Simulation auf mehrere Rechner verteilt. Die Eigenschaften pulsverarbeitender neuronaler Netze – insbesondere die Verzögerung zwischen Neuronen – werden auch bei der verteilten Simulation genutzt um eine effiziente, dezentrale, konservative Synchronisation der einzelnen Rechner zu erreichen. Dadurch gelingt es selbst noch im ungünstigsten Fall, nämlich der Verteilung eines vollständig vernetzten Netzwerks auf mehreren Rechnern, eine Beschleunigung durch die Verteilung nachzuweisen. Ein quantitativer Vergleich mehrer Nachrichtenübertragungsverfahren zeigt, dass CORBA als Nachrichtensystem zur Verteilung dieser ereignisgetriebenen Simulation besonders geeignet ist.

Schließlich zeigt die in dieser Arbeit entwickelte Beschleunigungshardware RACER in welcher Weise eine ereignisgetriebene Simulation durch Hardware beschleunigt und wie analoge und digitale Hardware in die Simulation eingebunden werden kann. Die RACER-Hardware ist eine lange PCI-Erweiterungskarte, welche drei austauschbare, programmierbare Modulen mit einem einfachen Datenfluss orientierten Interface beherbergt. Trotz der Kapazitätsbeschränkung des auf dem Prototyp-Modul eingesetzten FPGA, konnte eine Beschleunigung einzelner Teile der ereignisgetriebenen Simulation zeigt werden. Darüber hinaus wird durch die angestellten Messungen deutlich, dass mit komplexeren programmierbaren Bausteinen, welche sich durch das Modulkonzept leicht auf der bestehenden Hardware einsetzten lassen und auf denen sich ganze Subsimulatoren implementieren lassen, eine Verteilung der Simulation wie im Workstationcluster, jedoch mit einer deutlich leistungsfähigeren Kommunikation und optimierten Berechnungsknoten realisieren ließe.