Johann Wolfgang Goethe-Universität
Frankfurt am Main

Fachbereich Informatik (20) Praktische Informatik *VSFT*

Parallelarbeit und Fehlertoleranz in Multiprozessorsystemen

Dr.R.Brause, Bad Soden

Inhaltsverzeichnis

2. Parallelarbeit in Multi-Mikroprozessor-Systemen

2.1 Speichergekoppelte Systeme
Pluribus, C.mmp, NYU

2.2 Nachrichtengekoppelte Systeme
LAN Systeme, Cm*, Die Connection machine

2.3 Das ATTEMPTO-System
2.3.1 Anforderungen an das System
2.3.2 Die Hardwarearchitektur
2.3.3 Das Prozeßsystem
Die Benutzersicht von ATTEMPTO
Systemübersicht
Umleitung der System-Calls
Die Fehlertoleranzschicht FTL
Die Kommunikationsinstanz CI
Die Fehlertoleranzinstanz FTI
Die Datenpuffer
Die Zeitschranken
Die Resourcenverwaltung RM

2.3.4 Inter-Prozessor Koordination und Synchronisation
Konsistenz der Systemtafeln
Nachrichtenaustausch
Koordination des Nachrichtenaustauschs
Flußkontrolle und Fehlertoleranz
Zusammenfassung

2.3.5 Parallelisierungskonzepte
ATTEMPTO 1: Das Dispatching der Benutzerprogramme
ATTEMPTO 2: Das Parallelisierungskonzept
Das Implementierungskonzept

2.3.6 Simulation und Emulation des Systems
Die Prozeßstruktur der Simulation

2.3.7 Implementierung des Systems
Hardware
Software

3. Fehlertoleranz in Multi-Mikroprozessorsystemen

Konzepte der Fehlertoleranzmaßnahmen

3.1 Hardware-Konzepte
FTMP, Stratus

3.2 Software-Konzepte
SIFT, Tandem

Vorwort

Die vorliegende Arbeit entstand auf der Basis der Tätigkeit des Autors in verschiedenen Projekten an der Universität Tübingen und der Universität Frankfurt. Die Arbeit ist in vier Teile gegliedert und spiegelt dabei auch die Entwicklung der Forschungsinteressen des Autors unter den gemeinsamen Leitlinien *Parallelarbeit und Fehlertoleranz* wieder.

Dazu wird im ersten Teil von den systematischen Grundlagen der Parallelarbeit und Fehlertoleranz sowie den Arbeiten zu Test und Diagnose ausgegangen, die im zweiten und dritten Teil bei der Beschreibung von existierenden Multiprozessorsystemen referiert werden. Ein wichtiges Problem ist dabei der Gegensatz zwischen einerseits den stark gekoppelten Multiprozessoren, bei denen Zugriffsprobleme (*bus contention*) auftreten können, wenn für die gemeinsamen, globalen Daten ein gemeinsamer Speicher eingesetzt wird, und den lose gekoppelten Multiprozessoren andererseits, die lokal ohne Zugriffsprobleme effektiv arbeiten können, die globalen Daten allerdings nur mit einem relativ großen Zusatzaufwand (*Protokolle*) konsistent bearbeiten können.

Im zweiten Teil nehmen nach einigen Beispielen von existierenden Multiprozessor-Parallelrechnern einen besonderen Raum die konkreten Realisierungsmöglichkeiten der vorher skizzierten Probleme am Beispiel des experimentellen, fehlertoleranten Multimikroprozessorsystems ATTEMPTO ein, in dessen Arbeitsgruppe der Verfasser seit Projektbeginn mitarbeitet. Es wird gezeigt, wie in ATTEMPTO das Problem der konsistenten, dezentral gehaltenen, globalen Systemdaten elegant und fehlertolerant mit Hilfe eines speziellen Kommunikationsprotokolls gelöst werden kann. Das Konzept des dafür notwendigen *atomaren Rundspruchs (atomic broadcast)* wird für das Multi-Mikroprozessorsystem diskutiert und die Implementierung mit Hilfe handelsüblicher Bauteile vorgestellt.

Im dritten Teil wird dann näher auf die konkreten Möglichkeiten eingegangen, Fehlertoleranz in Multiprozessorsystemen zu verwirklichen. Nach einer Übersicht über Ansätze, die entweder überwiegend Hardware oder überwiegend Software dafür einsetzen, werden die Probleme geschildert, die sich zusätzlich in Prozeßsteuersystemen mit Real-Time Anforderungen ergeben. Danach wird der Software-orientierte Ansatz von ATTEMPTO beschrieben, der durch eine Kombination von Fehlermaskierungstechniken, Testgraphen und Signaturanalyse eine Fehlertoleranz für die Ausgabe in Programmen verwirklicht.
Obwohl über die Implementierung des ATTEMPTO 1 Konzepts und seiner Probleme bereits auch eine Weiterentwicklung zum Konzept von ATTEMPTO 2 vorgestellt wird, die stärker an einer "feineren Körnung" von Parallelarbeit und Fehlertoleranz orientiert ist, reicht dies nach Meinung des Autors für die Realisierung künftiger, hochkomplexer Maschinen der "künstlichen Intelligenz" nicht aus.

Aus diesem Grunde werden im vierten Teil massiv parallele, inhärent fehlertolerante Modelle der neuen Forschungsrichtung *Neuroinformatik* vorgestellt. Nach einer kurzen Einführung in das Gebiet wird am Beispiel der Modellklasse der "topologie-erhaltenden Abbildungen" die Möglichkeiten und Probleme der zentralisierten und der vollparallelen Version des Modellalgorithmus für die Anwendung in der Roboterkontrolle gezeigt. Danach werden die Fehlertoleranzeigenschaften der neuronalen Netze am Grundmodell des assoziativen Speichers demonstriert. Den Abschluß der Arbeit bilden Betrachtungen der Simulation derartiger Modelle mit den heutigen, zur Verfügung stehenden Mitteln, wobei die Überlegungen am Beispiel des von der Arbeitsgruppe des Autors entwickelte Simulationssystem INES konkretisiert werden.

An dieser Stelle möchte ich auch zuallererst Herrn Prof. Dr. M. Dal Cin danken, der mir über die Jahre hinweg zuerst in Tübingen und dann in Frankfurt die Gelegenheit gegeben hatte, die in dieser Arbeit beschriebenen Gedanken und Beiträge zu entwickeln.
Desweiteren richtet sich mein Dank an alle Mitglieder der Arbeitsgruppen von ATTEMPTO 1 und ATTEMPTO 2, insbesondere an E. Dilger, W. Günther, J. Lutz, Th. Philipp und Th. Risse für die fruchtbaren Diskussionen um Konzepte und Mechanismen.

Frankfurt am Main, im Frühjahr 1990

R. Brause

1. Grundbegriffe und Mechanismen

Für unsere Untersuchung komplexer Systeme ist es wichtig, zuerst die prinzipielle Funktion der Einzelteile und dann das Schema des Zusammenwirkens dieser Einzelteile zu betrachten. Deshalb soll zunächst im ersten Teil dieser Arbeit in die Thematik der parallelen und fehlertoleranten Maschinen eingeführt und dann Grundzüge der parallelen Hardwarearchitekturen und der parallelen Programmierung besprochen werden. Danach werden einige grundlegende Modelle vorgestellt, um die Tests, Diagnose und Reparatur dieser Maschinen zu systematisieren und in Richtung einer Toleranz gegenüber auftretenden Fehlern zu automatisieren.

1.1 Einleitung

Seit dem Erscheinen des Mikroprozessors in den 70-ger Jahren hat eine stürmische Entwicklung der Rechnertechnologie stattgefunden. Die enorme, stetige Verbesserung des Preis-Leistungsverhältnisses der verwendeten Elektronik ("Hardware") ließ den Einsatz von Rechnern auch in solchen Bereichen wie Handwerk und Büro sinnvoll und wirtschaftlich erscheinen, an die noch in den Zeiten millionenteurer Großrechner niemand zu denken gewagt hatte.
Dabei zeigte sich aber bald ein interessantes Phänomen: Je mehr Rechenleistung mit fortschreitender Technologie für das gleiche Geld angeboten werden konnte, umso mehr verlangten die Benutzer. Wie wird nun die zusätzliche Rechenleistung erreicht und wie wird sie verwendet?

von Neumann Rechner vs. parallele Architekturen
Eine höhere Rechenleistung läßt sich prinzipiell auf zwei verschiedene Arten erreichen: zum einen durch den Einsatz einer verbesserten Technologie bei der Realisierung bestehender Rechnerarchitekturen (z.B. ECL anstelle von TTL) und zum anderen durch den Entwurf neuer Architekturen mit bestehender Technologie. Obwohl der klassische Monoprozessor mit seiner von Neumann Architektur (sequentielle Bearbeitung von Befehlen) parallel arbeitenden Architekturen wie beispielsweise Multiprozessoren in der Leistung bei gleicher Technologie unterlegen ist, ist trotzdem die Verbreitung von Multiprozessoranlagen ziemlich gering.
Der Grund dafür liegt in der fortschreitenden Fertigungstechnologie. Ist ein Multiprozessorsystem auf der Grundlage eines Monoprozessors nach 2 jähriger Entwicklungszeit endlich marktreif, so ist inzwischen bereits der kompatible Nachfolgetyp des Prozessors mit 2-10 facher Leistung erschienen. Ein Anwender, vor die Alternative gestellt, entweder sein Programm völlig neu für eine parallele Abarbeitung zu formulieren (mit allen Folgefehlern und Serviceproblemen!) oder aber durch Einsatz eines neuen Prozessors sein Programm zu beschleunigen, wird natürlich das letztere wählen.
Da sich mittelfristig bereits weitere Technologieverbesserungen wie GaAs-Technik und Quanten-Tunneleffekttransistoren mit Schaltzeiten von Bruchteilen von Pikosekunden angekündigt haben, werden auch mittelfristig die Monoprozessoren stark genutzt werden. Dies schließt die Nutzung paralleler Architekturprinzipien (*pipelining, Coprozessoren*) auch in Monoprozessor-Computersystemen (s. Kapitel 1.3.1) mit ein.

Trotzdem ist es aber sinnvoll, die Erforschung und Entwicklung der Multiprozessorsysteme parallel dazu weiterzutreiben. Dies resultiert nicht nur aus der Überlegung, daß die rasante Technologieentwicklung der Fertigung irgend einmal zu Ende sein könnte, wie sich dies beispielsweise heutzutage bei der Nutzung konventioneller Masken-und Ätztechniken bei Strukturen kleiner als 1 Mikrometer zeigt, sondern auch aus den Fortschritten im Herstellungsprozeß (CIM) und der Programmierung der Computersysteme. Gelingt es, ein effektives Modell eines parallelen Rechners zu entwickeln, es an einem Prototyp zu validieren und geeignete Entwicklungswerkzeugen für parallele Programme dafür zur Verfügung zu stellen, so bietet sich die Möglichkeit, einmal entwickelte parallele Architekturen und Programme mit kurzem Zeitverzug zum Erscheinen des Monoprozessors zur Verfügung zu stellen. Die so erzielten Leistungssteigerungen lassen sich nur durch ungleich teurere Technologien (und damit Computer) erzielen. Deshalb haben alle größeren Computerfirmen heutzutage auch Parallelrechner in ihrem Angebot, die sich aus Benutzersicht allerdings aus den oben genannten Gründen wie Monoprozessorsysteme verhalten. Um die Leistung paralleler arbeitender Prozessoren nicht nur auf Systemebene (z.B. Concurrent Computer Systems: Pro Benutzer ein Prozessor), sondern auch auf Programmebene nutzen zu können, muß auch das vom Benutzer geschriebene Programm (der Programmalgorithmus) die Multiprozessor-architektur ausreichend berücksichtigen. Obwohl viele derartige, parallel zu programmierende Maschinen kommerziell bereits ein Erfolg geworden sind, bleibt das große Problem der effektiven, parallelen Anwendungsprogrammierung nach wie vor bestehen.

Die Bereitstellung benutzerfreundlicher, schneller Rechner verlangt von der Informatik große Anstrengungen, die Grundprinzipien der Werkzeuge zur effektiven, parallelen Programmierung bereitzustellen, die zusätzlich auch noch den fehlertoleranten Einsatz ermöglichen soll.
Was sind die Hauptprobleme dieser Werkzeuge ?

Benutzerschnittstellen

Spätestens, seitdem auch völlig ungeübte Büroangestellte ohne Vorbildung mit der Bedienung ihrer Text- und Dokumentenverarbeitungssysteme konfrontiert werden, ist klar, daß eine der wichtigsten Vorbedingungen für die Einführung von Computern im Firmenalltag die Akzeptanz durch ihre Benutzer ist. Dies ist aber ohne eine gute, den Menschen angepaßte Benutzerschnittstelle nicht möglich. Der höhere Leistungsbedarf ist damit nicht nur auf den Einsatz der Mikroprozessor-basierten Rechner in bisher ungenutzte Bereiche mit höherer Mindestleistung (CAD,etc) zu erklären, sondern vor allem in der Verbesserung der Benutzerschnittstelle durch rechenintensive, graphische Hilfsmittel (Visualisierung!), um die Benutzerführung zu vereinfachen.

Ein wesentliches Kriterium einer guten Benutzerschnittstelle ist nicht nur die Benutzung ergonomischer Visualisierungsmittel wie Ikone, Fenster und Zeigeinstrumente ("Mäuse"), sondern auch die Vermeidung von Fehlern durch Beachtung des Kontextes, der Benutzergewohnheiten, mit einem Wort: der Fehlertoleranz in den Benutzerschnittstellen. Dabei weisen die Neuronalen Netze, wie im drittenTeil näher ausgeführt wird, besonders interessante Proportionen auf.

Effektive Programmierung

Ein weiteres großes Problem der stark um sich greifenden Nutzung von Rechnern ist deren Programmierung. Konnte es sich IBM in den 60-ger Jahren noch leisten, das Betriebssystem und die Systemprogramme großzügig den Computerkäufern dazu zu schenken, so drehte sich das Kostenverhältnis bald um; gute CAD-Programme für Architekturbüros beispielsweise kosten gut das doppelte bis fünffache der reinen Gerätekosten der Rechenanlage. Da sich das Denk-Tempo der Menschen nicht einfach so steigern läßt wie beispielsweise die Taktfrequenz der Prozessoren, bildet heutzutage - trotz der Entwicklung von Software-Managementmethoden und Programmier-Hilfssystemen wie z.B. die Sprachen der 4. Generation und die analytischen Programmentwicklungsmethoden (CASE) - für ein gegebenes Anwenderproblem im Regelfall die Programmentwicklung das größte Problem und nicht die Hardware.

Da aber nach Statistiken der amerikanischen Armee ca. 90% der Programmierarbeiten dieses Benutzerkreises auf Programmpflegearbeiten entfallen und nicht auf Neuentwicklung, erweist sich der bisherige Ansatz der Informatik, eine Maschine zu benutzen, der alles genau bis ins kleinste Detail beschrieben werden muß, was sie tun soll, als sehr problematisch.

Im dritten Teil dieser Arbeit wird deshalb diskutiert, wie sensorische und motorische Systeme der "Künstlichen Intelligenz "(KI), die auch in normalen Betrieben Verwendung finden sollen, auch ohne direkte Programmierung durch Eingabe von Trainingsmustern (Beispielen) ihre Aufgabe lernen können. Beispielsweise konnte die gleiche Aufgabe, ein Computersystem auf Spracherkennung zu programmieren, die von einem Entwicklungsteam von Digital Equipment in 20 Mann-Jahren mit 95% Genauigkeit erreicht wurde, durch ein einfaches, simuliertes Netzwerk nur durch die Eingabe der Testworte in 16 Lernstunden mit einer Wiedererkennungsgenauigkeit von 98% gelöst werden (Sejnowski:NETalk).

Parallele Programmierung

Ein wichtiger Aspekt der Programmierung ist die Schwierigkeit, die zur Leistungssteigerung zu Multiprozessorsystemen zusammengekoppelten Monoprozessoren auch effektiv und fehlerfrei zu programmieren. Sieht man von den stark vernetzten, parallelen Systemen der neuronalen Netze (des dritten Teils) einmal ab, so ist die fehlerfreie Programmierung der vernetzten Multiprozessorsysteme auf automatische Werkzeuge angewiesen, die die Komplexität der Hardware dem Programmierer verbirgt und trotzdem eine effektive, leistungsteigernde, parallele Programmierung der Aufgaben gestattet. Dies ist einführenderweise im ersten Teil und im Detail im zweiten Teil am Beispiel des ATTEMPTO Systems behandelt.

Hardware-Fehlertoleranz

Im Laufe der Jahre sind die Rechner immer komplexer und zuverlässiger geworden. Die Alltagserfahrung, daß nämlich Ausfälle umso wahrscheinlicher sind, je größer und komplexer ein System ist, scheint sich für Rechner nicht bestätigt zu haben. Der Grund dafür mag in der zunehmenden Integrationsdichte aber auch in der zunehmenden Perfektionierung der Herstellungsprozesse und in der Perfektionierung der Test- und Ausleseverfahren zu finden sein. Je komplexer aber die Systeme werden, desto eher stößt ihre Perfektionierung an Machbarkeits- und Rentabilitätsgrenzen. Deshab wurden in der Rechnertechnik schon frühzeitig Verfahren entwickelt, die dafür sorgen, daß Rechner trotz Auftretens einzelner Fehler funktionstüchtig

bleiben.

Dabei stellten sich aber rasch neue Probleme ein. Das eine Problem ist eng an den technischen Fortschritt gekoppelt: der mögliche Ausfall des Rechnersystems und seine Folgen für den Benutzer. Gerade durch die Integration der elektronischen Komponenten mit immer größeren Dichten auf bis zu einer Million Transistorfunktionen pro Chip steigt durch steigende Transitfrequenz der aktiven Bauelemente und geringere Parasitärkapazitäten einerseits die mögliche Taktfrequenz und damit der Datendurchsatz durch den Chip, andererseits aber auch die Wahrscheinlichkeit, daß der Chip mit dem Ausfall einer Transistorfunktion unzuverlässig arbeitet. In normal benutzten Systemen versucht man durch den Einbau von funktionsbegleitenden Tests (Prüfung der Quersumme bei Speicherelementen, Massenspeichern und Datenbussen) und speziellen Testprozeduren beim Einschalten des Systems (Start-Up Test) den Fehlern zu begegnen. Ist dies bei einem Monoprozessorsystem noch ausreichend, so ist bei einem Rechnersystem aus mehreren Prozessoren für den ungeübten Benutzer dies nicht mehr so einfach möglich. Hier, sowie bei besonders ausfallsicheren, lebenserhaltenden Systemen, müssen noch andere Maßnahmen ergriffen werden, über die in den folgenden Kapiteln besonders gesprochen werden wird. Als Beispiele seien fehlerkorrigierende Codierung, robuste Kommunikations-Protokolle, testfreundliche Schaltungsentwürfe und gegenseitige Funktionsüberwachung bei integrierten Bausteinen genannt. Solche Verfahren garantieren in bestimmtem Umfang die Ausfallsicherheit auch bei Fehlern, d.h. sie bewirken Fehlertoleranz.

Zählt Fehlertoleranz zu den dominierenden Architekturkriterien eines Rechners, so spricht man von einem *fehlertoleranten Rechner.* Ein fehlertoleranter Rechner ist zuverlässiger als die Gesamtheit seiner Teile. Daß in letzter Zeit immer häufiger von solchen Rechnern die Rede ist, liegt vor allem daran, daß Rechner immer weiter in Anwendungsgebiete vordringen, die den Rechnern ein Höchstmaß an Zuverlässigkeit oder Verfügbarkeit abverlangen. Solche Bereiche finden sich z.B. in der Luft- und Raumfahrttechnik und der Medizin, aber auch in der Telekommunikation, im Bankwesen, im Handel oder in der Fertigung. Forst & Sullivan berichten, daß mit einem stetig zunehmenden Anteil der fehlertoleranten Systeme an europäischen DV-Umsätzen zu rechnen ist. Während sie 1986 nur ein Prozent des Hardwareumsatzes ausmachten, sollen es 1992 bereits sieben Prozent sein. Auf den europäischen Markt entfallen 25 Prozent des gesamten Absatzes an fehlertoleranter Hardware. Zunehmende Bedeutung gewinnt der Einsatz von Fehlertoleranztechniken in verteilten Systemen (Rechnernetzen) und massiv parallelen Rechnerarchitekturen, die in ihrer Struktur über ein allgemein nutzbares Redundanzreservoir verfügen. Die kommerzielle Verfügbarkeit fehlertoleranter Systeme wird in den kommenden Jahren viele Anwendungen überhaupt erst möglich machen.

Es liegt auf der Hand, daß die Fehlertoleranz dort, wo ein Versagen des Rechners mit großen, wirtschaftlichen Verlusten oder gar mit der Gefährdung menschlichen Lebens verbunden ist, eine besondere Bedeutung zukommt. Aber auch schon die Einsicht, daß ein Rechnerversagen das Vertrauen der Kunden in die Zuverlässigkeit der eigenen Firma entscheidend mindern könnte, läßt an fehlertolerante Rechner denken. Zuweilen begegnet man auch dem Argument, daß fehlertolerante Systeme trotz der erforderlichen Redundanz und der zusätzlichen Funktionalität bei gleicher Zuverlässigkeit kostengünstiger sein könnten als eine

perfekte Spezialentwicklung, da das fehlertolerante System ja aus preiswerten Standardkomponenten aufgebaut werden könnte. Dabei ist allerdings zu bedenken, daß die Entscheidung zugunsten einer fehlertoleranten Architektur und eines bestimmten Fehlertoleranzkonzepts schon früh im Prozeß der Systementwicklung getroffen werden muß, denn ein Aufsetzen der Fehlertoleranz im nachhinein wird selten zu einem befriedigenden Ergebnis führen. Dazu sind die Mechanismen, die die Fehlertoleranz einbringen sollen, viel zu sehr mit dem Gesamtverhalten des Systems verwoben.

1.2 Modelle paralleler Architekturen

Die Fülle der existierenden informationsverarbeitenden Maschinen heutzutage gestattet hier keine ausführliche Beschreibung konkreter Maschinenarchitekturen, sondern nur noch die Unterscheidung nach Architekturtypen, besser: Typen von Architekturelementen.
Folgen wir der populären Einteilung von Flynn [FLY], so wird an einem Problem entweder mit einem Maschinenbefehlsstrom (**S**ingle **I**nstruction) oder mit mehreren (**M**ultiple **I**nstruction) gearbeitet. Dabei wird ein Datenstrom (**S**ingle **D**ata) oder mehrere Datenströme (**M**ultiple **D**ata) bearbeitet. Entsprechend den Abkürzungen der englischen Bezeichnung wird ein einzelner Prozessor, der mit einer einzelnen Instruktion eine Operation auf einem Speicher ausführt, als SISD-Rechner bezeichnet. Wird eine Operation gleichzeitig auf vielen Daten ausgeführt wie beispielsweise in Feldrechnern, so ist die Bezeichnung SIMD üblich. Bei vielen Operationen auf einem Datenstrom (MISD) sind fließbandartige Konfigurationen gegeben, während mit MIMD sich schließlich die Multiprozessorsysteme beschreiben lassen, die wir im Folgenden betrachten werden.

Multicomputer- und Multiprozessorsysteme
Die vier möglichen Abkürzungskombinationen deuten aber nur sehr grob die Merkmale der Rechnerarchitekturen an. Deshalb sind zusätzliche Kennzeichnungsmerkmale üblich, die das Zusammenwirken der Prozessoren charakterisieren lassen.

Statten wir einen Prozessor (SISD) mit Speicher und Peripherie aus und koppeln wir mehrere derartiger Computer zu einem lokalen Netz (LAN) zusammen, so wird dies als Multicomputersystem bezeichnet. Da in einem solchen System meist jeder Computer unterschiedliche Aufgaben erledigt, wird es als *verteiltes System* angesehen; die Computer kommunizieren nur mittels Nachrichten, deren Senden und Empfangen einen zusätzlichen Aufwand (*Overhead*) bedeuten.

Im Gegensatz dazu bearbeiten die meist in einem Gehäuse integrierten Multiprozessorsysteme gleichzeitig eine einzige Aufgabe und werden deshalb als *paralleles System* bezeichnet (s. [BHU]). Dabei wird klassischerweise zwischen *eng gekoppelten* (*tightly coupled*) Systemen unterschieden, bei denen ein einheitlicher, physikalisch meist an einer Stelle befindlicher, gemeinsamer Speicher (*common, shared memory*) existiert und den *lose gekoppelten* (*loosely coupled*) Systemen, bei denen der von allen Prozessoren adressierbare Speicher physikalisch aufgeteilt jeweils einem Prozessor zugeordnet (und von diesem lokal schnell erreichbar) ist. Bei den lose gekoppelten Prozessoren gibt es also eine Speicherzugriffshierarchie: lokale Zugriffe benötigen deutlich weniger Zeit als Zugriffe auf Zellen des globalen Speichers, die meist nur über spezielle Adressierungsmechanismen indirekt erreicht werden können.

Diese klassiche Aufteilung ist allerdings nicht unproblematisch. Es gibt viele Systeme, die hardware- oder softwaremäßig eine Zwischenstellung zwischen beiden Extremen einnehmen. Im Folgenden soll dies etwas genauer diskutiert werden.

Bei eng gekoppelten Systemen wird versucht, die Zugriffszeiten auf den gemeinsamen Speicher durch Einführung eines zusätzlichen, lokalen Speichers (*Cache*) zu senken. Als wichtiges Unterscheidungsmerkmal zu den lose gekoppelten Systemen bleibt noch die Nutzung des lokalen Speichers: Der Cacheinhalt ist beim eng gekoppelten System immer ein Abbild des im gemeinsamen Speicher vorhandenen Programms (*Instruktionscache*) oder seiner Daten (*Datencache*). Wählen wir allerdings die Cache-Speicher so groß, daß ganze, abgeschlossene Programmteile (z.B. Prozesse) darin Platz finden und schnell abgearbeitet werden können und verteilen wir im lose gekoppelten System die Programmteile ebenso auf die lokalen Speicherbereiche, so wird auch dieses Unterscheidungsmerkmal bedeutungslos. Verwenden wir außerdem für den asynchronen Datenaustausch zwischen den Prozessen ein einheitliches Datenformat (*messages*), so verblassen auch die funktionellen Unterschiede zu den Multicomputersystemen.

Haben nun eng gekoppelte Syteme generell eine *schnellere* Verbindung zwischen den Prozessoren?

Nein, denn auch der Zeit- und Geschwindigkeitsaspekt (Kommunikationsbandbreite) ist bei neueren Systemen schwer als Klassifizierungsmerkmal zu gebrauchen: ein moderner *packet--switched* Multi-Prozessor Bus (enge Kopplung) erreicht im Unterschied zu den traditionellen Bussen (UNIBUS, VME-Bus) seine hohe Bandbreite durch die Zusammenfassung der Daten zu einzelnen Datenpaketen, also im Prinzip durch Nachrichten (messages). Trotz ähnlicher Funktionalität kann er dabei aber sogar langsamer sein als eine schnelle Lichtleiterkopplung im Multicomputersystem (sehr lose Kopplung).

Was ist nun als charakteristisches Unterscheidungsmerkmal noch zu verwenden?

Bei beiden Systemen existiert ein Kommunikationsprotokoll, das zur Kommunikation intelligente Arbiter, Datenpuffer und dergleichen benötigt, so daß auf den ersten Blick keine prinzipiellen zeitlichen oder funktionalen Architekturunterschiede zwischen Multicomputersystemen und Multiprozessorsystemen vorhanden sein müssen.

Trotzdem gibt es aber einen wichtigen Unterschied. Betrachten wir vor allem bei den LAN-Systemen die zusätzlichen Anforderungen an die Kommunikationsschicht (logisch/physikalische Adressierung etc, siehe Kapitel 1.4.2) der Multicomputersysteme. Hier fällt ein prinzipiell nötiger Mehraufwand für die Nachrichtenübermittlung ins Gewicht. Dabei muß das Anwenderprogramm, bedingt durch das Kommunikationsmodell, allerdings nicht zwischen dem Multicomputersystem und dem lose gekoppelten Multiprozessorsystem unterscheiden.

Anders ist dagegen die Situation beim fest gekoppelten Multiprozessorsystem. Hier lassen sich im Unterschied zu den nachrichtengekoppelten Systemen sehr unproblematisch auch globale Variable (s. Kapitel 1.3.3) implementieren, so daß diese Architekturgruppe für bestimmte Programmiersprachen und Datenmodelle (z.B. bei Datenbanken) besonders gut geeignet ist.

Die Systeme lassen sich also durch ein wichtiges Merkmal trennen, das primär nicht durch das Architekturmodell, sondern durch das Programmiermodell gegeben ist: das Merkmal der *speicherorientierten* oder der *nachrichtenorientierten Kopplung*. Die meisten Hardware-Architekturen sind für die Implementierung keiner der beiden Programmiermodelle problemlos zu verwenden, so daß "reine" Systeme, in denen nur ein Programmiermodell auf allen Ebenen Verwendung findet, nur selten anzutreffen sind.

Kopplung in Multiprozessorsystemen

Der Hauptunterschied in der Programmierung (und damit auch in der Benutzersicht) zwischen den verschiedenen Architekturen ist also die Behandlung globaler Daten. Da der gleichzeitige Zugriff von vielen Prozessoren auf solche Daten (und Programme) problematisch ist, unterscheiden sich die Rechnerarchitekturen hauptsächlich in dem Grundansatz, dieses Problem befriedigend zu lösen. Die Methode dabei läßt sich nach [ALM] in zwei Ansätze aufteilen: den "Tanzsaal" Ansatz, bei dem jeder Prozessor sich ein Speicher-"Partner" wählen kann, und dem "Vorzimmer"-Ansatz, bei dem der Zugang zu einem Prozessor/Speicher-Paar über einen gemeinsamen Kommunikationsweg ("Vorzimmer") erfolgt, s. Abbildung 1.2.1.

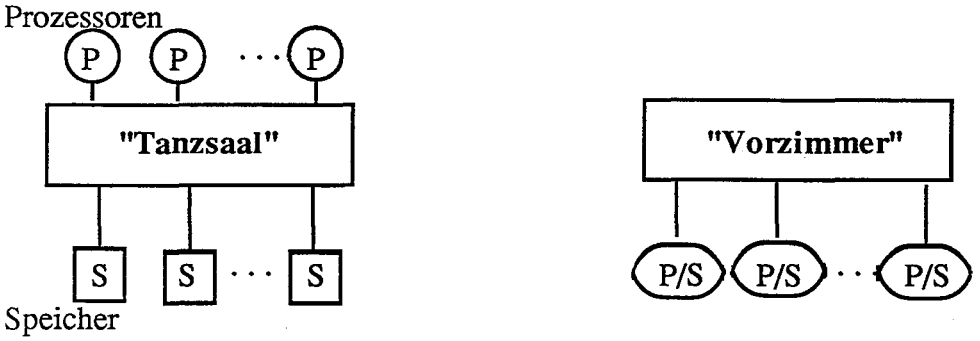


Abb. 1.2.1 "Tanzsaal" und "Vorzimmer" Architekturen

Hierbei wird der "Tanzsaal"-Ansatz eher für feste, speicherorientierte Kopplungen und der "Vorzimmer"-Ansatz eher für lose, nachrichtenorientierte Kopplungen verwendet.

Welche Verbindungsarchitekturen sind nun als "Tanzsäle" oder "Vorzimmer" geeignet?

Die zweifelsohne einfachste und nebenwirkungsfreieste Verbindungsart besteht darin, alle Komponenten mit allen anderen direkt zu verbinden (volle Vernetzung). Dies bedeutet aber einen großen Harwareaufwand (N·(N-1) Verbindungen!) und ist nur schwer erweiterbar.

Verwebt man die Kommunikationsstruktur fest mit den Prozessoren und Speichern, um einen Algorithmus möglichst schnell zu bearbeiten, so erhält man regelmäßige Strukturen wie Bäume und Gitterstrukturen (*Systolische Felder, s. Abb. 1.5.3*).

Das andere Extrem besteht darin, mit einem sehr niedrigen Vernetzungsgrad alle Komponenten in einer Kette, in einem Ring oder in sternförmiger Art miteinander zu verbinden. Diese Strukturen werden durch den indirekten Datenaustausch meist für nachrichtenorientierte Systeme, z.B. für LANs, verwendet.

Die tatsächlich genutzten Multiprozessorsysteme verwenden meist Verbindungsarten, die in dem Aufwand zwischen der vollen Vernetzung und der Linienkette liegen. Betrachten wir im Folgenden eine Auswahl der gebräuchlichsten Netztopologien.

Nachdem in den 60-er Jahren das Architekturmerkmal, alle Komponenten des Rechnersystems durch dedizierte Hardwarekanäle (z.B. Leitungen) zu verbinden, zugunsten eines einzigen, einheitlichen, leicht erweiterbaren Kommunikationsweges (*Bus*) aufgegeben wurde, dominiert diese Architektur auch bei Multiprozessorsystemen (*Multi-Master-Bus*), siehe Abbildung 1.2.2.

Allerdings hat diese Lösung einige Nachteile. Die beiden wichtigsten Probleme sind einerseits die geringe Fehlertoleranz beim Ausfall des einzigen Kommunikationsweges und andererseits die (relativ zu den angeschlossenen Prozessoren) meist geringe Kommunikationsbandbreite, so daß schon bei zwei bis drei Prozessoren sich Sättigungseffekte (*Flaschenhals*) bemerkbar machen

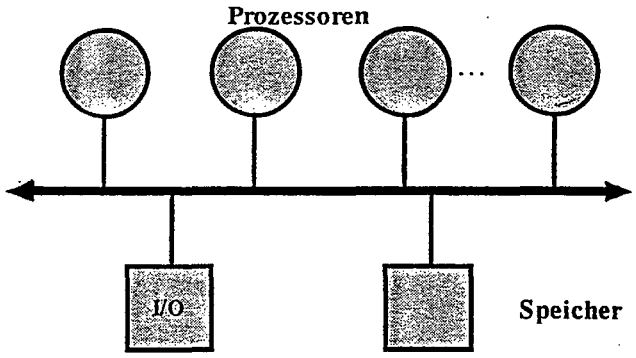


Abb. 1.2.2 Single bus, multi-master System

können. Auch der Einsatz einer schnelleren Technologie beim Bussystem ändert daran nicht sehr viel, da schnellere Technologien meist zuerst schon bei der Prozessorfertigung eingesetzt werden und damit beim Bussystem wieder das gleiche Problem auftritt. Abhilfe schafft für diese Probleme ein Kommunikationssystem, bei dem mehrere Wege zwischen den Prozessoren und den Speichereinheiten existieren. Das einfachste Beispiel ist ein multiples Bussystem, wie es in Abbildung 1.2.3 vorgestellt wird.

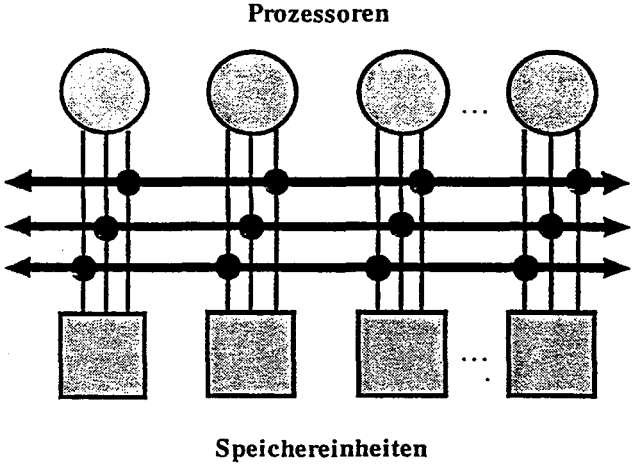


Abb. 1.2.3 Multiple bus, multi master System

Allerdings ist es fraglich, ob bei vielen Datenzugriffen dieses Zugriffssystem ausreicht. Fordert man nun eine besonders schnelle Kopplung zwischen allen Prozessoren und allen Speichereinheiten, so ist sicher der Kreuzschienenverteiler (s.Abb. 1.2.4) die kompromißloseste Lösung. Hier stören sich die Zugriffe der verschiedenen Prozessoren nicht, sofern sie nicht dieselbe Einheit addressieren. Ein Beispiel dafür ist das C.mmp System, das in Abschnitt 2.1 näher beschrieben ist.

Die Auslastung dieses Systems ist weitgehend durch die geschickte Verteilung der verwendeten Softwareteile (abgeschlossene Objekte) auf die Speichermodule und die Beschränkung der Datenzugriffe auf ebenso abgeschlossene Bereiche gegeben (Statistik der Speicherzugriffe!). Der Preis, der dafür gezahlt werden muß, ist allerdings ziemlich hoch: Bei N Prozessoren und M Speichern sind M*N Schalter notwendig, was bei mehreren hundert Prozessoren und Speichereinheiten zu Technologieproblemen führen kann.

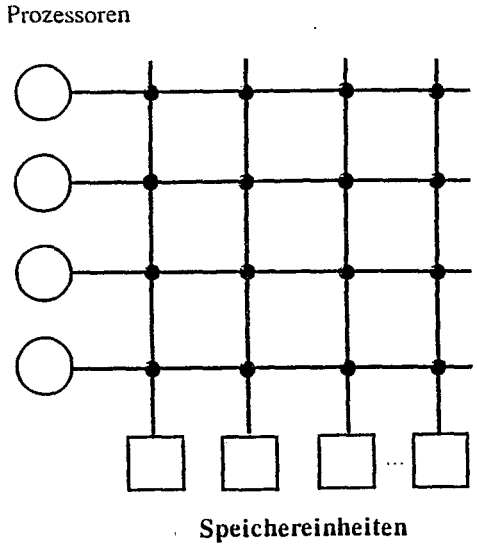


Abb. 1.2.4 Kopplung durch Kreuzschienenverteiler

Verteilt man stattdessen die Schaltfunktionen auf mehrere, sequentiell zu durchlaufende Stufen (*Multi-stage Interconnection Network, MIN*), so läßt sich die Zahl der nötigen Schalter stark reduzieren. In Abbildung 1.2.5 ist ein "Omega-Netzwerk" mit M=N=8 zu sehen, das Schalterelemente mit 2 Eingängen und 2 Ausgängen (2x2Schalter) verwendet. Jeder Vermittlungswunsch trägt zusätzlich Kontrollbits mit sich (3 Bit in Abb.1.2.5), die sequentiell die Schalter schließen. Damit läßt sich ein mehrstufiges Netzwerk verteilt aufbauen und kontrollieren; Erweiterungen, beispielsweise für Fehlertoleranz, sind durch Kaskadierung leicht möglich. Ein Beispiel dafür ist der NYU Ultracomputer in Abschnitt 2.1.

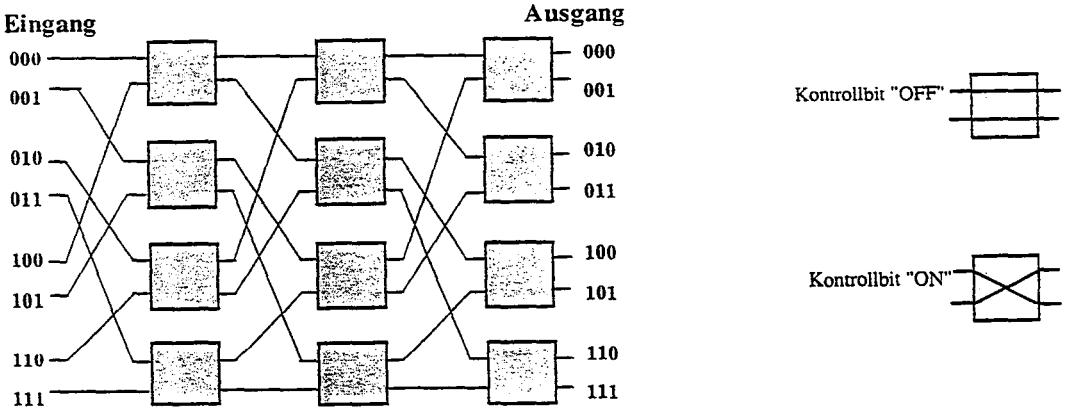


Abb.1.2.5 Ein 8x8 Omega Netzwerk

Andere Beispiele für mehrstufige Netzwerke sind "delta" Netze ($N=a^n$, $M=b^n$, n Stufen mit a x b Schaltern), generalisierte "shuffle" Netzwerke ($N=n_1n_2 \ldots n_r$, $M= m_1m_2 \ldots m_r$, r Stufen mit n_i x m_i Schaltern), Banyan-Netzwerke usw. Als ein besonderes Beispiel sei noch die Hauptlinie der Hypercube-Architekturen (cube-connected cycle etc) erwähnt, bei der jedes Prozessor/Speicher-Paar ein Eckpunkt in einem n-dimensionalen, bool'schen Würfel bildet (s. Abb.1.2.6). Hier hat jeder Prozessor Zugang zu den *n* Nachbarn aller 2^n Prozessoren. Fällt eine Buskontroll-Einheit aus,

so gibt es auch noch andere Wege über andere Ecken des Würfels, um zu einem bestimmten Prozessor zu kommen. Ein Beispiel dafür ist die Connection Machine aus Abschnitt 2.2.
Für einen ausführlicheren Überblick sei auf [FEN] verwiesen.

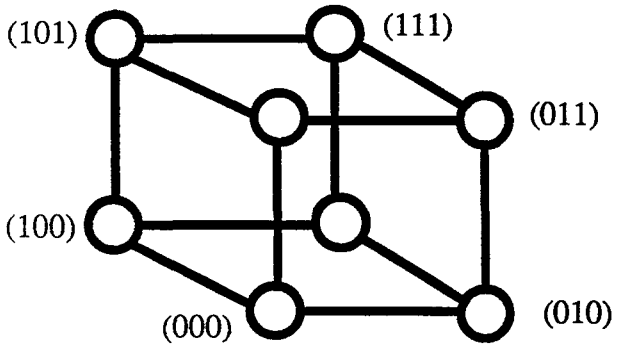


Abb. 1.2.6 Hypercube für n=3

Diskussion

Vergleicht man nun die drei vorgestellten Kopplungs-Architekturprinzipien des Kreuzschienenverteilers, der Mehrstufenschalter MIN und der multiplen Bussysteme in Bezug auf Zuverlässigkeit, Rekonfigurierbarkeit und Erweiterbarkeit, so bietet sich mit dem geringsten Hardwareaufwand (und damit geringster Ausfallwahrscheinlichkeit) zuerst das multiple Bussystem an. Aber auch ein Performance-Vergleich, wie er von Yang [YAN] für paketvermittelte Bussysteme durchgeführt wurde, zeigte gute Ergebnisse. Beispielsweise erreichte ein System mit 4 Bussen bei 16 Prozessoren und 16 Speichereinheiten fast die gleiche Prozessorauslastung wie der schnelle, aber wesentlich aufwendigere Kreuzschienenverteiler. Dies ist wohl hauptsächlich darauf zurückzuführen, daß die zusätzlichen, parallelen Verbindungsmöglichkeiten des Kreuzschienenverteilers normalerweise von der Software gar nicht ausgenutzt werden (können).

Da die Kontrolle, Fehlertoleranz, Rekonfigurierbarkeit und Erweiterbarkeit bei dem Kreuzschienenverteiler (und besonders bei den Mehrstufenverbindungsnetzwerken) sehr problematisch und schwierig durchzuführen sind, bietet sich für die Multiprozessorsysteme besonders die Netzwerkarchitektur der multiplen Busse an.

1.3 Modelle paralleler Aktivierung

Die Entwicklung der preiswerten Mikroprozessoren brachte über Spezialanwendungen hinaus die Frage an die Entwickler von Programmen, Programmsprachen und Hardwaresystemen: Wie können die neuen Hardwaremöglichkeiten sinnvoll für die Beschleunigung bestehender und zukünftiger Programme genutzt werden ?
Nach Ghezzi [GHE] müssen alle parallelen Programme Folgendes leisten:

- *Bezeichne eine Menge von Aufgaben, die parallel ausgeführt werden können*
- *Starte und beende ihre Ausführung*
- *Spezifiziere und koordiniere ihre Interaktion während der Ausführung*

In den folgenden Abschnitten betrachten wir deshalb die grundsätzlichen Möglichkeiten, parallele Aktionen sprachlich zu spezifizieren, anzustoßen und zu beenden sowie die Probleme, solche parallel ablaufenden Aktionen auf Single- und Multiprozessoren zu koordinieren.

1.3.1 Parallele Sprachkonstrukte

Der größte Anreiz für die Bemühungen um parallele Sprachkonstrukte besteht zweifelsohne in einer gewünschten, schnelleren Abarbeitung der Programme. Der einfachste und sicherlich auch preiswerteste Weg dahin besteht allerdings nicht etwa darin, eine neue, parallele Programmiersprache zu erlernen, sich für jedes Anwenderproblem neue, parallele Algorithmen zu überlegen und die vorhandenen Programme komplett umzuschreiben, sondern darin, ein einziges Mal einen "intelligenten" Compiler zu schreiben, der alle *implizite* Parallelität in den Programmen ausnutzt und einen Code produziert, der auf parallel arbeitender Hardware ablaufen kann.

Leider ist dieser universelle Ansatz nach den bisherigen Erfahrungen nur begrenzt brauchbar. Im Normalfall läuft ein komplett neu konzipierter, paralleler Algorithmus wesentlich schneller ab als ein "automatisch parallelisierter", sequentiell konzipierter Algorithmus. Der Grund dafür liegt weitgehend in dem komplizierten, kreativen Prozeß zum Ableiten eines solchen Algorithmus, der sich sehr schwer automatisieren läßt.

Betrachten wir im Folgenden die sprachlichen Konstrukte näher, um *explizit* parallele Aktivitäten zu beschreiben. Dabei konzentrieren wir uns auf die verschiedenen, expliziten sprachlichen Ausdrucksmöglichkeiten für potentielle Parallelität innerhalb von Programmen in den konventionellen, imperativen Programmiersprachen, die stark die von-Neumann-Struktur der verwendeten Computer widerspiegeln und hardwarenah zur Implementierung der parallelen Aktivitäten anderer Sprachtypen, beispielsweise der objektorientierten, funktionalen oder logischen Programmiersprachen, benutzt werden können.

"doall"-Konstrukte
Eine der ersten Versuche bestand darin, Programmierschleifen, in denen voneinander unabhängige Datenobjekte bearbeitet wurden, als Parallelkonstrukte zu beschreiben. Dies ist die Familie der *doall* Konstrukte [ALM, p.155], zu denen auch *forall, pardo* und *doacross* gehören. Ein Beispiel mit dem "forall" Konstrukt ist in Abbildung 1.3.1 zu sehen.

```
FOR i=1 TO M
    FOR j=1 TO M
       FOR k=1 TO M
         a[i,j,k] = n*b[i,j]
        END
      END
    END
```

```
Forall i,j,k inparallel
    a[i,j,k] = n*b[i,j]
END
```

Abb. 1.3.1 Sequentielle und parallele Formulierung von Schleifen

In dem Beispiel werden die voneinander unabhängigen Operationen, die links in der Abbildung als drei geschachtelte, sequentielle Schleifen formuliert werden, in der Abbildung rechts als einheitliches Konstrukt formuliert. Es obliegt dann dem Compiler, aus den potentiell parallelen Aktionen durch eine adäquate Abbildung auf die tatsächlich vorhandenen Hardwaremöglichkeiten einen parallelen (oder sequentiellen) Ablauf zu erzielen.
Wichtig bei dem obigen Beispiel ist die Tatsache, daß die parallelen Aktivitäten an einem bestimmten Punkt (END) alle abgeschlossen wieder in das sequentielle Programm einmünden müssen. Dieser Abschlußpunkt wird als *Barriere* bezeichnet; die Synchronisation der parallelen Aktivitäten heißt *Barrierensynchronisation* (*barrier synchronisation*) nach H. Jordan [JOR].
Man beachte, daß die echt parallele, unabhängige Ausführung der Schleifen nur dann möglich ist, wenn die Datenobjekte (hier: Feldelemente) nicht von anderen, in anderen Schleifen errechneten Daten (beispielsweise von einem Feldelement mit kleinerem Index) abhängig sind. In diesem Fall muß der Programmierer selbst entscheiden, inwieweit die parallele Formulierung gleiche Ergebnisse wie die sequentielle bringt. Beachtet man die Abhängigkeit der Daten voneinander und zeichnet diese in Form eines gerichteten Graphen auf, wobei die Daten als Knoten und die Abhängigkeit der Bearbeitungsreihenfolge mit Kanten dargestellt wird, so erhält man einen Datenabhängigkeitsgraphen (*Datenflußgraphen*). Spezielle Sprachen wie VAL, LAU oder SISAL, die Sprachkonstrukte für die Datenabhängigkeiten enthalten (*Datenflußsprachen*), sowie die Maschinen, die sie ausführen können (*Datenflußmaschinen*), werden aber hier nicht näher behandelt. Stattdessen sei auf die Übersicht [AGE] verwiesen.

"fork/join"- Konstrukte

Eine der bekanntesten Syntaxkonstrukte, um parallel ablaufbare Aktionen zu starten und wieder zu beenden, ist das *fork/join* Konstrukt. Dabei wird mit 'fork' notiert, welche Aktivität (Prozeß, Prozedur oder Befehlssequenz) gestartet und bei 'join', welche beendet wird. In Abbildung 1.3.2 ist ein Beispiel gezeigt, das die Markierungen der Befehlsequenzen benutzt; der Programmcode ist links und der Kontrollfluß ist rechts im Bild eingezeichnet.
Es ist deutlich zu sehen, daß durch die direkte Zuweisung von Abspaltung und Zusammenführung der Programmaktivitäten der Kontrollfluß dem eines unstrukturierten "goto"-Konstrukts entspricht; man kann deshalb das "fork/join"-Konstrukt als ein "Multiprozessor-goto" bezeichnen. Der Programmcode, der mit Hilfe dieser Konstrukte geschrieben wird, hat deshalb die typischen Mängel der "goto"-Konstruktionen: Er ist schwer zu verstehen, unübersichtlich und damit fehleranfällig und schwer zu warten (*Spaghetti-Code*). Anderseits sind die Operationen so elementar, daß mit diesen Konstrukten leicht höhere, wohlstrukturierte Syntaxkonstrukte implementiert werden können.

```
        ...
        fork A
        ...
        fork B
        ...
        join A
        ...
        join B
        ...
        exit
A:      ...
        exit
B:      ...
        exit
```

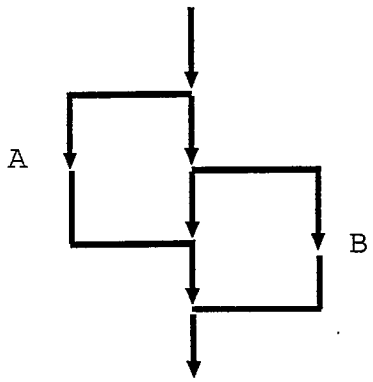


Abb. 1.3.2 Beispiel für einen **fork** und **join** Kontrollfluß

Die Bedeutung des 'join' Kommandos ist die einer Barriere; der bezeichnete Prozess muß an dieser Stelle im Programm anhalten und sich mit den anderen Prozessen bzw. dem Hauptprogramm koordinieren.

parbegin/parend Konstrukte

Im Unterschied zu den elementaren fork/join Konstrukten erlauben die strukturierten *parbegin/parend* Konstrukte (z.B. cobegin/coend) nur das Aufsetzen und Anhalten aller parallelen Aktivitäten innerhalb des mit 'parbegin/parend' markierten Blocks. Damit ist dieses Konstrukt als Spezialfall der fork/join Konstruktionen restriktiver in der Anwendung als fork/join, aber auch überschaubarer. In Abbildung 1.3.3 ist ein Beispiel gezeigt.

```
parbegin
  ...
  A
  B
  ...
parend
```

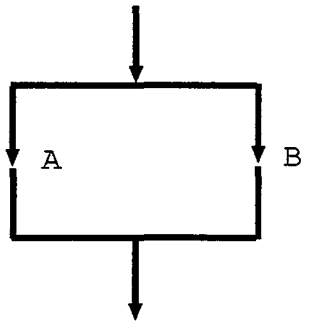


Abb. 1.3.3 Beispiel für einen **parbegin/parend** Kontrollfluß

Dabei können die parallelen Aktivitäten `A` und `B` ebenfalls wieder Prozesse, Prozeduren oder nur einfache Befehle sein. Das `parend` ist bei diesem Konstrukt eine Barriere für alle Prozesse.

1.3.2 Parallelität in Single-Prozessorsystemen

Obwohl das Modell des Von-Neumann-Rechners nur sequentielle Aktionen kennt, wird bei der Realisierung in konventionellen Computersystemen bereits vielfach die Parallelität ausgenutzt, die physikalische Systeme von Natur aus bieten.
Dies geschieht im Wesentlichen auf folgenden Ebenen (nach Kober [KOB])

- *Job- oder Programmebene*
  Die von verschiedenen Benutzern konzipierten Programme sind meist unabhängig voneinander und könnten damit zeitparallel auf verschiedenen Single-Prozessorsystemen ausgeführt werden.

- *Coroutinen und Prozeßebene*
  Voneinander unabhängige Programmteile können als eigenständige Einheiten ausgeführt werden. Muß eine Einheit auf ein externes Ereignis warten (z.B. Peripheriedaten), so kann "parallel" dazu eine andere Einheit vom Prozessor bearbeitet werden.

- *Instruktions- und Expressionsebene*
  Hier können voneinander unabhängige Daten und Instruktionen, von "intelligenten" Compilern gut aufbereitet, parallel bearbeitet werden. Beispiele dafür sind Feldrechner (SIMD), Datenflußmaschinen und Pipeline-Architekturen bei Prozessoren. Die parallele Ausführung wird hierbei durch koordinierte, parallel arbeitende Teile des Prozessors unterstützt.

- *Bitebene*
  Die parallele Verarbeitung mehrerer Bits (z. B. 16, 32, 64 Bits) im gesamten System sowie die parallele Abarbeitung der Maschineninstruktionen im Prozessor (z.B. RISC Design) ermöglicht eine schnellere Bearbeitung bei unverändertem Programm.

In den heutigen Arbeitsplatzcomputern wird die Abarbeitungsgeschwindigkeit der Programme durch eine Kombination obiger Techniken auf allen Ebenen verkürzt. Angefangen von einer großen Wortbreite im System über On-Chip Daten- und Instruktionscache, pipe-lining und paralleler Befehlsdecodierung im Prozessor (SIMD-ähnlich!), getrennte, parallel arbeitende Adressübersetzung (On-Chip MMU) für Instruktionen und Daten (z.B. im Motorola 68040) und Fließkomma-Coprozessoren bis hin zu "intelligenten" Peripheriecontrollern, die selbst wieder kleine Computersysteme mit CPU, RAM, ROM, Peripheriechips und eigenem Betriebssystem sind, reichen die Maßnahmen zur Beschleunigung der Programmausführung.

Granularität

Die parallel ablaufenden Befehlsabschnitte dauern auf den verschiedenen Ebenen verschieden lang; von Nanosekunden auf der Bitebene bis zu Stunden auf der Jobebene. Diese unterschiedlich lang dauernden, parallelen Aktivitäten sind meist auch mit unterschiedlich langen Befehlssequenzen gekoppelt, so daß man die parallelen Aktivitäten auch mit unterschiedlich großen "Körnern" vergleichen kann; man spricht von der *Körnigkeit* oder *Granularität* der betrachteten Aktivitäten. Parallelität auf der Instruktionsebene wird als *feinkörnig* angesehen im Unterschied zu der *grobkörnigen* Parallelität von parallel abgearbeiteten Jobs.

Diese Unterscheidung ist nicht nur für echt parallel arbeitende Multiprozessorsysteme wichtig, sondern auch für Single-Prozessorsysteme. Jedes Umschalten zwischen Jobs oder Programmteilen erfordert nämlich einen Verwaltungsaufwand zum Abspeichern des alten und Restaurieren des neuen Zustands. Ist auch das Betriebssystem bei der Umschaltung (z.B. von Jobs und *Prozessen*) beteiligt, so ist außer dem Prozessorstatus (Registerinhalte) auch der Status im Betriebssystem (Speicher- und Filedeskriptoren etc) zu konservieren. Wird innerhalb eines Prozesses nur zwischen verschiedenen Programmteilen umgeschaltet, so spricht man wegen des

geringeren Aufwandes von *Leichtgewichtsprozessen* oder *Coprozessen*, wobei der Aufwandsunterschied zu den Systemprozessen bis zu mehreren Zehnerpotenzen betragen kann.

Multiprogramming und Interrupts

Man kann im Single-Prozessor-System die parallel zum Hauptprogramm (*nebenläufig*) ablaufende Aktivität eines Zeitgebers (Timer) dazu benutzen, die Abarbeitung des Programms periodisch zu unterbrechen. Wird dabei vom Betriebssystem über eine Neuvergabe (*Scheduling* und *Dispatching*) des Prozessors für den nächsten Zeitabschnitt (*Zeitscheibe*) entschieden, so ist mit diesem Mechanismus eine "gleichzeitige" Bearbeitung mehrerer Programme (*Multitasking*) oder der Jobs mehrerer Benutzer (*Multiprogramming*) möglich. In Real-Time-Systemen wird die Unterbrechung eher durch externe Ereignisse verursacht; die Neuvergabe des Prozessors wird dabei auf der Basis von Prioritäten entschieden.
In beiden Fällen treten allerdings schon mit den Unterbrechungen (*Interrupts*) durch parallele, unabhängige Ereignisse die für Multiprozessorsysteme typischen Probleme auf, die hier kurz erläutert werden sollen.

Atomare Aktionen, kritische Abschnitte und Semaphoren

Die Interrupts wirken sich besonders bei der Bearbeitung globaler, von mehreren Programmteilen benutzten Datenstrukturen aus. Wird beim Zugriff auf eine solche Datenstruktur für ihre Konsistenz eine nicht-unterbrechbare Sequenz von Befehlen (*atomic action*) eines Programmabschnitts (*critical region*) verlangt, so bewirkt jedes Unterbrechen einer solchen atomaren Aktion und Fortfahren eines anderen Programms auf den selben Daten eine Dateninkonsistenz und damit einen Fehler. Beispiele hierfür sind das Aushängen oder Einfügen von Elementen einer verzeigerten Liste sowie das Hinzufügen oder Entnehmen eines Elements aus einem Datenpool mit der notwendigen Aktualisierung eines Zählers.

In Single-Prozessor Systemen wird dieses Problem dadurch gelöst, daß man beim Eintritt in einen kritischen Abschnitt eine binäre Markierung (*Semaphore*) setzt. Ruft ein anderes Programm eine Prozedur auf, die den selben kritischen Abschnitt (z.B. im Betriebssystem) benutzt, so wird zuerst die Semaphore abgefragt. Ist sie schon gesetzt, so wird das laufende Programm als Prozeß blockiert und gibt die Prozessorkontrolle ab. Hat ein Prozeß den kritischen Abschnitt verlassen, so setzt er die Semaphore zurück und weckt alle Prozesse auf, die blockiert waren. Der erste der aufgeweckten Prozesse, der den kritischen Abschnitt unblockiert vorfindet und die Semaphore gesetzt hat, bekommt die Kontrolle und bewirkt eine erneute Blockierung aller anderen. Benutzt man anstelle der binären Semaphoren nicht-binäre Semaphoren (z.B. ganze Zahlen), so kann die Semaphore die Zahl der blockierten Prozesse widerspiegeln.

Da die Abfrage und das Setzen bzw. Erhöhen der Semaphore selbst wieder als kritischer Abschnitt nicht erneut durch eine Semaphore abgesichert sein kann, benutzt man üblicherweise für diese atomare Aktion spezielle, hardwareunterstützte Funktionen (z.B. *Test-and-Set*), die ununterbrechbar abgearbeitet werden. Die Abfrage und das Blockieren des Prozesses wird nach Dijkstra [DIJ] als *P-Operation*, das Deblockieren und Aufwecken als *V-Operation* zusammengefaßt. Die Gesamtheit von P- und V-Operationen, kritischem Code und anderen Operationen für die globalen Daten sowie den globalen Daten selbst kann man zu einem *Monitor* zusammenfassen und damit als *gekapseltes Objekt* ansehen. Der Zugriff auf die globalen Daten geschieht dann nur noch exklusiv über den Monitor und garantiert so konsistente Daten.

1.3.3 Parallelität in Multi-Prozessorsystemen

In Multiprozessorsystemen lassen sich grundsätzlich zwei verschiedene Organisationsformen der zeitparallelen (*nebenläufigen*) Datenbearbeitung unterscheiden: Die Bearbeitung als parallelen Datenfluß und die Bearbeitung im sequentiellen Datenfluß, siehe Abbildung.

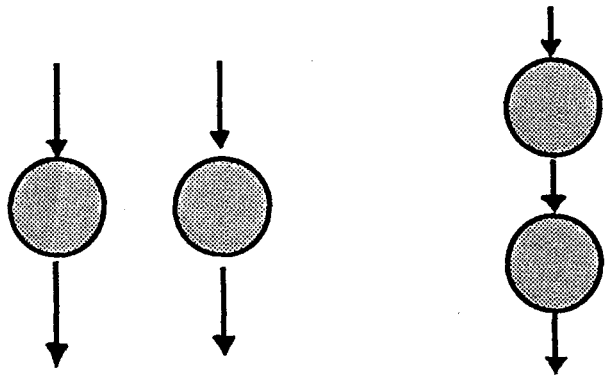


Abb. 1.3.4 Paralleler und sequentieller Datenfluß

Der sequentielle Datenfluß wird auch in Analogie zu Röhrensystemen als *pipeline* bezeichnet. Sind mehrere pipelines parallel geschaltet, spricht man von *systolischen Feldern*. Unabhängig von der Hardwarearchitektur lassen sich diese Bearbeitungsmodelle sowohl mit speichergekoppelten als auch mit nachrichtengekoppelten Systemen (s.u.) durchführen.

Bei der parallelen Bearbeitung mehrerer Programmteile auf einem Multiprozessorsystem entstehen nun ähnliche Probleme, wie sie bereits im vorigen Abschnitt erörtert worden sind. Dabei sind es vor allem zwei Problemkreise, die hier zufriedenstellend gelöst werden müssen:

- *Laufzeitsynchronisierung paralleler Prozesse*
  Für eine Interaktion zwischen parallel ablaufenden Prozessen, die ohne gegenseitige Blockierung (Deadlock) ablaufen soll, sind gewisse Synchronisationspunkte sehr sinnvoll. Diese Punkte (Barrieren) müssen so implementiert werden, daß möglichst wenig Prozessorleistung verloren geht.

- *Zugriffskoordinierung paralleler Prozesse*
  Der Zugriff der echt parallel ablaufenden Prozesse auf globale Resourcen, beispielsweise auf globale Datenstrukturen, muß wie bei den nur versetzt abgearbeiteten, sequentiellen Prozessen koordiniert werden. Die im vorigen Abschnitt beschriebenen Mechanismen eignen sich aber nicht unbedingt auch für Multiprozessorsysteme.

Bei der Erörterung der Mechanismen, die für eine Synchronisierung oder Koordinierung nötig und möglich sind, müssen wir allerdings zwischen den beiden Hauptarchitekturen der Rechner (s. Abschnitt 1.2) unterscheiden. Generell läßt sich sagen, daß auf eng gekoppelten Rechnern Algorithmen schneller abgearbeitet werden, die globale Daten verwenden und eng synchron ablaufen müssen. Demgegenüber vermeiden Algorithmen auf lose gekoppelten Systemen, die ohne dauernde Aktualisierung globaler Daten auskommen, das Problem der Zugriffsüberlastung (*bus*

contention) auf oft benutzte Speicherstellen (*Flaschenhals*). Benutzt man Modelle der beiden Architekturen und zerlegt bekannte, parallele Algorithmen in Einzelschritte je nach Architektur, so lassen sich über die Laufzeitkomplexitäten die Eignung der Architekturen für bestimmte Algorithmen miteinander vergleichen. In Gottlieb und Kruskal [GOTT] ist dies für Probleme der Permutation, Mittelwertwertbildung, Sortierung etc. durchgeführt worden.
Betrachten wir nun im Folgenden die Grundmechanismen und -probleme, die sich auf beiden Architekturen für die Laufzeitsynchronisierung und Zugriffskoordinierung ergeben.

Speichergekoppelte Systeme

In diesen Rechnersystemen mit gemeinsamem globalen Speicher läßt sich im Allgemeinen eine Laufzeitsynchronisierung leicht durch einen Barriere erreichen, die mittels einer Zugriffskoordinierung auf globale Resourcen implementiert wird, so daß die Zugriffskoordinierung bei diesen Systemen als Hauptproblem erscheint.

Zugriffskoordinierung und Daten-Cache

Ein typisches Hardwareproblem der speichergekoppelten Probleme ergibt sich, falls ein Cache-Hilfsspeicher (s. Abschnitt 1.2) bei jedem Prozessor (z.B. bereits auf dem Prozessor-Chip!) eingeführt wird. Dabei besteht nicht nur die Notwendigkeit, benötigte, aber nicht vorhandene Instruktionen beim Instruktions-Cache aus dem Hauptspeicher nachzuladen, sondern beim Daten-Cache die veränderten Daten auch wieder konsistent zurückzuspeichern.
Es gibt verschiedene Möglichkeiten, die Dateninkonsistenzen in den globalen Daten zu verhindern. Eine Alternative besteht darin, Hardwaremechanismen einzubauen, die bei Datenänderungen auch automatisch die Daten sowohl im Hauptspeicher als auch in den anderen Daten-Cache der anderen Prozessoren zu aktualisieren (*broadcast update* [YEN]) oder sie mindestens als "ungültig" zu markieren. Diese *cross invalidation* ist beispielsweise in den IBM 370/168 und IBM 3081 Maschinen zuerst verwendet worden.
Eine andere, software-orientierte Möglichkeit, die Cache-typischen Probleme der Zugriffskontrolle auf globale Daten zu umgehen, besteht in besonderen Deklarationsformen für globale, *non-cachable* Daten. Können Compiler und Linker diese Anweisungen verarbeiten, so läßt sich ein Hardware-Mechanismus einrichten, der das Laden derartiger Daten in den Cache verhindert. Das Problem dieses Ansatzes liegt allerdings darin, daß das Cache-Problem auf den Programmierer abgewälzt wird. Vergißt er eine Variable, so können die sich daraus ergebenden Probleme äußerst schwer diagnostiziert und lokalisiert werden. Aus Gründen der Fehlervermeidung ist deshalb meist eine Hardwarelösung statt einer Softwarelösung installiert. Ein Überblick über die Cache-Probleme ist in [SMITH] gegeben.

Zugriffskoordinierung im Hauptspeicher

Der exklusive Zugriff auf globale Resourcen läßt sich durch den mit Semaphoren gesicherten Zugriff auf globale Daten implementieren. Allerdings ist dies bei einer großen Anzahl von konkurrierenden Prozessen nicht sehr ratsam, da durch das notwendige Nacheinanderausführen der ursprünglich gleichzeitigen Zugriffe auf dieselbe Speicherstelle (test-and-set derselben Semaphore) die Zugriffszeiten sehr verlängert werden. Außerdem ist jeder Zugriff als atomare Aktion auch ununterbrechbar, was zusätzliche Wartezeiten abverlangt.

Ein weiterer kritischer Punkt ist der automatische Prozeßwechsel, der bei gesetzter Semaphore

in der P-Operation auftritt und immer mit einem Overhead verbunden ist. Auch wenn damit nur ein Prozess und nicht ein Prozessor blockiert wird, lassen sich andere Strategien überlegen, die diese Nachteile vermeiden.
Eine Alternative besteht darin, die test-and-set-Operation nur für eine Abfrage zu benutzen, im Falle eines Mißerfolgs aber etwas anderes zu unternehmen (berechnen etc), ohne den Prozeß zu blockieren, und danach erneut abzufragen. Geschieht die Zusatzaktivität nur durch einen Prozeduraufruf oder einen Leichtgewichtsprozeß (s. Abschnitt 1.3.2), so läßt sich der Overhead stark reduzieren.

Eine andere Möglichkeit, die Blockierung der test-and-set Operation gerade für den Zugriff von sehr vielen Prozessoren aufzuheben, ist die Synchronisierungsprimitive *fetch-and-add(S,n)*. Analog zu der test-and-set Primitive liest sie eine Speicherstelle S und addiert dann noch die Zahl n in einem atomaren Schritt dazu. Obwohl sich test-and-set als binäre Variante von fetch-and-add ansehen läßt, sind doch die Möglichkeiten verschieden. Falls die Zugriffsoperation durch Hardware im Netz unterstützt wird wie beim NYU Supercomputer (s.Abschnitt 2.1), kann ohne Zeitverlust die Operation von allen Prozessoren auf der selben Speicherstelle im selben Zyklus durchgeführt werden. Als Ergebnis erhalten alle Prozessoren verschiedene Zahlen, die von ihnen interpretiert (s.o.) werden können.

Nachrichtengekoppelte Systeme
Die einfachste Aktion für die Laufzeitsynchronisierung und Zugriffskoordination in diesen nachrichtengekoppelten Systemen ist das Senden und Empfangen von Nachrichten.

Laufzeitsynchronisierung
Eine Barriere für die Synchronisierung parallel ablaufender Prozesse läßt sich leicht durch das Warten auf die "Fertig"-Nachrichten der fraglichen Prozesse implementieren. Ein wichtiges Beispiel dafür ist die Abarbeitung einer Prozedur auf einem anderen, parallel dazu arbeitenden Prozessor. Dazu sendet der beauftragende Prozessor (*client*) eine Auftragsnachricht mit den Prozedurargumenten an den anderen Prozessor (*server*), der die gewünschte Dienstleistung erbringt (die Prozedur abarbeitet) und die Ergebnisse in einer Nachricht zurücksendet.
In der synchronen Form dieses Mechanismus wartet der Auftraggeber (client) blockiert auf die Ergebnisnachricht und arbeitet dann weiter (*Remote Procedure Call, RPC*); in der asynchronen Form arbeitet der client nach Auftragsvergabe sofort weiter und holt sich erst zu einem späteren Zeitpunkt (Barriere) das zwischenzeitlich eingetroffene Ergebnis aus einem Nachrichtenempfangspuffer (*Remote Service Invocation, RSI*).

Ermöglicht man den RPC oder RSI beim client über Prozeduren mit gleicher Syntax (Laufzeitbibliothek), die aber unsichtbar für den Programmierer das Versenden und Empfangen der Argumente und Ergebnisse übernehmen (*Stub-Prozeduren*), so fügt sich diese Form der parallelen Aktivierung (besonders beim blockierenden RPC ohne explizite Barrieren im client) für den Programmierer problemlos und überschaubar in die Sprachmittel der jeweiligen prozeduralen Programmiersprache ein. Besonders in Netzwerken (LAN) ist der RPC für die gemeinsame Nutzung von Resourcen beliebt; bei inhomogenen Netzen mit Prozessoren und Betriebssystemen verschiedener Hersteller müssen jedoch bei der Auftragsvergabe auch die Typen und Formate der verwendeten Argumente rechnerunabhängig beschrieben werden (ISO Norm 8825, Basic Encoding Rules). In Abbildung 1.3.5 sind die Softwareschichten eines solchen RPC gezeigt.

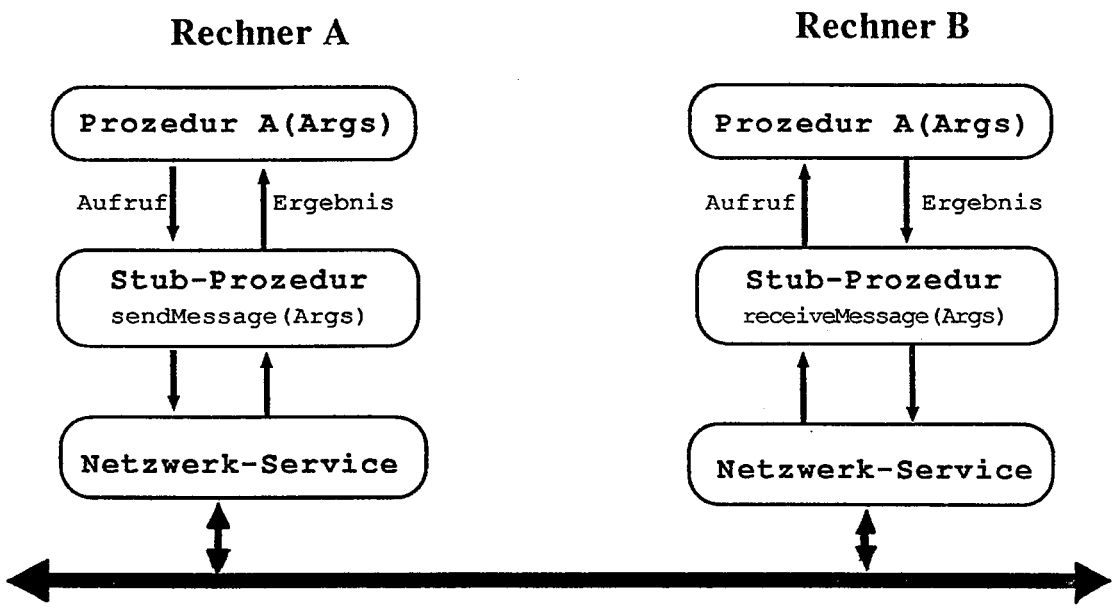


Abb.1.3.5 Aufrufschichten eines Remote Procedure Call

Dabei kann in der Netzwerkschicht auch die Existenz einer Einheit verborgen sein, die aus dem symbolischen Namen des Beauftragten die aktuelle Adresse im Netzwerk des zu beauftragenden Prozessors ermittelt (*Name-Server*).

In homogenen Systemen mit gleichartigen Prozessoren stellt ein RPC im Unterschied zum RSI allerdings im Prinzip kein Zeitgewinn dar, da der Auftraggeber während der Ausführung des Auftrags blockiert bleibt und damit den Auftrag ohne Nachrichtenübermittlungs- und Verwaltungsaufwand schneller selbst erledigen könnte. Nur in inhomogenen Systemen, in denen bei einem Prozessor besondere Resourcen (Floating-Point Vektorprozessor, I/O ports, Massenspeicher etc) existieren, lohnt sich ein Auslagern der Aktivität, um diese Resourcen zu nutzen.

Zugriffskoordinierung

Im Unterschied zu der Laufzeitsynchronisierung ist allerdings die Zugriffskoordinierung in nachrichtengekoppelten Systemen relativ schwierig zu erreichen. Dabei lassen sich zwei Problemkreise unterscheiden:

Zum einen muß ein unverhältnismäßig großer Aufwand getrieben werden, um über Nachrichten einen einfachen, korrekten Zugriff auf globale Daten abzuwickeln: Dauert ein Zugriff in speichergekoppelten Systemen nur eine Bustaktperiode im Mikrosekundenbereich, so muß man in nachrichtengekoppelten Systemen mit einer mindestens 1000 fachen Zeit (Millisekundenbereich) rechnen. Dies führt im Bereich der verteilten Datenbanksysteme zu der Strategie, möglichst wenige globale Daten bei einem Zugriff zu sperren; für jede Teildatei wird eine extra Semaphore eingerichtet. Eine Transaktion belegt beim Zugriff in einer ersten Phase sukzessiv die Semaphoren und gibt sie in einer zweiten Phase nacheinander wieder frei (*Zwei-Phasen-Protokoll*), s. [BERN], beispielsweise durch eine Interaktion aller Prozessoren (*atomic commitment protocol*).

Zum anderen geraten auch diese wohlausgewogenen, knappen Protokolle leicht in problematische Zustände, falls Hardwareausfälle die Datenkonsistenz gefährden. Wichtig ist diese Problematik vor allem in verteilten Datenbanksystemen. Eine Lösungsmöglichkeit bietet die

Einführung einer *atomaren Transaktion*, bei der die Daten entweder konsistent geändert werden oder gar nicht.
Die Problematik eines fehlertolerant und konsistent aktualisierten, globalen Datenbereichs (*delta-common storage*) wurde von Cristian et alii in [CRIS1] behandelt. Sie gaben Kommunikationsprotokolle an, die mit Mindestannahmen über das verwendete Nachrichtensystem bei geringer Mehrbelastung solche Fehler wie Auslassung, Zeitfehler und Verfälschung (*Byzantinischer Fehler*) tolerieren und die selbe Nachrichtenreihenfolge bei allen Empfängern garantieren. Grundlage ihrer Überlegungen ist die Existenz eines sog. *Atomic Broadcast*, bei dem eine Nachricht in endlicher Zeit entweder alle Adressaten erreicht oder keinen, wobei die Nachrichtenreihenfolge überall gleich ist.

Die gleiche Reihenfolge der Nachrichten bei allen Empfängern läßt sich als nicht-blockierender Grundmechanismus einer konsistenten Datenhaltung verwenden. Sind überall Reihenfolge und Inhalt der Nachrichten gleich, so resultieren bei gleichem Anfangszustand und gleichen Änderungen auf allen Rechnern im Netz die gleichen Zustände der globalen Daten (vollst. Induktion). Diese Erkenntnis, die sich als grundlegend für die Datenkonsistenz [BERN ,pp.11] in den 80-ger Jahren vor allem im Bereich der verteilten Datenbanken etabliert hat, wurde unabhängig davon beispielsweise auch bei der Konzeption des ATTEMPTO-Systems (1982) entwickelt, um die Systemtafeln der globalen Daten (Resourcen etc) konsistent zu erhalten (s. Kapitel 2.3.4).

1.4 Mechanismen der Fehlertoleranz

Eine der Grundvorraussetzungen für Fehlertoleranz in einem System ist die Existenz von *Redundanz*. Hierbei kann so verschiedenes gemeint sein wie zusätzliche, für die eigentliche Funktion unnötige Hardware- oder Software- Komponenten oder die für die Fehlertoleranzzwecke benötigte, zusätzliche Systemzeit. Im allgemeinen wird jede Fehlertoleranzmaßnahme mit einer Mischung der verschiedenen Redundanzarten verwirklicht.
Allerdings muß dabei eines beachtet werden: Redundanz allein genügt nicht, um Systeme fehlertolerant zu machen. Vielmehr muß die vorhandene Redundanz auch zielgerichtet für Fehlertoleranz eingesetzt werden [BR13].

1.4.1 Redundanzarten

Betrachten wir im Folgenden die Stärken und Schwächen von Systemen, die sich fast ausschließlich auf eine der erwähnten Redundanzarten stützen. Für einen vollständigeren, ausführlicheren Überblick sei beispielsweise auf [AND], [DAL1] oder [GOE] verwiesen.

Zeitredundanz
Eine der einfachsten Maßnahmen, den Effekt von kurzzeitig aufgetretenen (*transienten*) Fehlern bei Ablauf eines Programms zu kompensieren, besteht in der (mehrmaligen) Wiederholung des Programms. Diese Möglichkeit bietet sich immer dann an, wenn Fehler nur selten auftreten, genügend Zeit für weitere Durchläufe zur Verfügung steht (*Zeitredundanz*) und das Programm nicht einmalige, wichtige Daten verarbeitet hat. Ist letzteres nicht mehr gegeben, wie beispielsweise in Transaktionssystemen, so müssen zur Sicherheit die Daten (Zustand des Programms) regelmäßig gespeichert werden. Tritt nun ein transienter Fehler auf, so wird das System in den letzten gespeicherten Zustand (*Rücksetzpunkt*) zurückversetzt (*Roll-back*) und beginnt erneut zu arbeiten. Die Befehlsfolge zwischen zwei Rücksetzpunkten wird dabei als atomare Aktion (s. Abschnitt 1.3.2) implementiert.

Schwierigkeiten bei diesem Fehlertoleranzmechanismus können entstehen, wenn für die Wiederholung einer atomaren Aktion auch der Zustand der Quelle der Eingabedaten zurückgesetzt werden muß. Ist dies ebenfalls ein Computersystem mit einem Rücksetzpunkt, so müssen die Entstehungszeitpunkte der beiden Rücksetzpunkte nicht gleich sein und es kann zu Dateninkonsistenzen kommen, die weiteres Rücksetzen anderer Computersysteme bewirken kann (*Domino-Effekt*).

Eine weitere Voraussetzung ist die Existenz von "gutartigen" Hardwarefehlern, also von Fehlern, die bei ihrem Auftreten ein "Fehler"-Signal auslösen sowie die entsprechende Komponente inaktivieren (*fail-stop*). Ist dieser inaktive Zustand auch für das Gesamtsystem unbedenklich, so kann man mit derartigen Komponenten sogenannte *fail-save* Systeme aufbauen. Die Zeitredundanz des Roll-back Verfahrens versagt nun genau dann, wenn Software-Fehler auftreten, Hardware-Fehler nicht zu "fail-stop" führen bzw. fatale Folgen für das Gesamtsystem haben (z.B. Ausfall der einzigen CPU) oder die Zeit für eine Wiederholung nicht zur Verfügung steht (z.B. in Real-time Systemen). Muß man in dem betrachteten System mit einem von diesen Fällen rechnen, so ist es besser, eines der folgenden Fehlertoleranzverfahren zu verwenden.

Hardwareredundanz

Die einfachste Möglichkeit, nach Ausfall einer Hardwarekomponente das System mit geringerer Leistung weiterarbeiten zu lassen (*graceful degradation*), besteht in der Verdoppelung aller wichtigen Komponenten. In Abbildung 1.4.1a ist eine beliebte Multiprozessor-Konfiguration, ein sogenanntes "Doppelprozessor" (*Duplex*) System gezeigt. Jeder Prozessor erhält seine Aufträge (beispielsweise Transaktionen) unabhängig vom anderen und benutzt dabei die ihm zugeordnete Magnetplatte. Außerdem hat er aber auch eine Zugriffsmöglichkeit auf die Platte des anderen Prozessors.

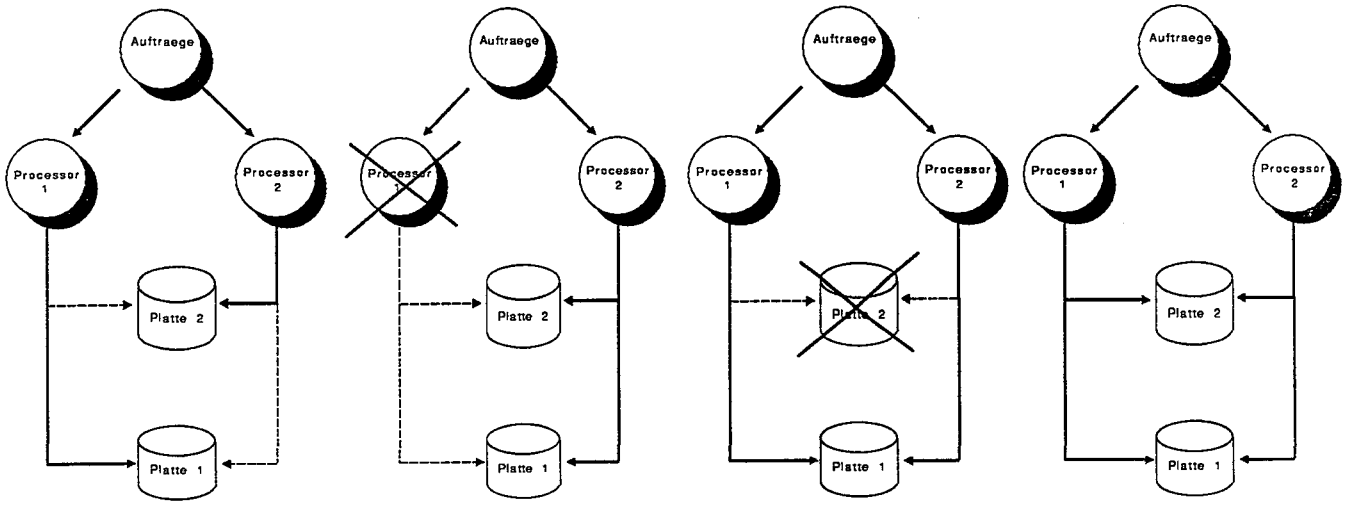


Abb. 1.4.1 a) **b)** **c)** **d)**

Fällt nun ein Prozessor (s.Abb.1.4.1b) oder eine Platte (s.Abb.1.4.1c) aus, so können nach Feststellen des Ausfalls (fail-stop!) und einer Rekonfiguration die verbleibenden Komponenten als System weiterarbeiten. Beim Ausfall einer Platte können nun Daten verloren gehen, die nur sehr schwer oder überhaupt nicht wiederbeschafft werden können. In Abbildung 1.4.1d ist ein Schema gezeigt, bei dem durch Ausführen der Aktionen auf beiden Platten jeweils gleiche Daten enthalten sind (*Spiegelplatten*). Der Vorteil der Fehlertoleranz bei Ausfall einer Magnetplatte wird dabei mit halbierter Speicherkapazität bezahlt, da jeder Prozessor nur auf der für ihn reservierten Plattenpartition arbeiten darf.

Eine wichtige Voraussetzung für die Arbeit eines Duplexsystems ist allerdings die ausschließliche Existenz von gutartigen Fehlern. Können dagegen auch Fehler auftreten, die nicht entdeckt werden und die Daten verfälschen, so muß zur Fehlerentdeckung parallel zum Betrieb der Betrieb selbst überprüft werden. Im Zweiprozessorsystem kann dies dadurch geschehen, daß beide Prozessoren das Gleiche tun und zusätzlich die Resultate ihrer Arbeit (z.B. in Form von Signaturen) miteinander vergleichen. Tritt nun ein Fehler bei einem der beiden Prozessoren auf, so kann das Zweiprozessorsystem nicht entscheiden, *wo*, sondern nur *daß* überhaupt ein Fehler aufgetreten ist. Die einzig sinnvolle Reaktion besteht dann darin, die Arbeit anzuhalten und das Problem einer übergeordneten Instanz (Operator) zu melden. Damit kann ein einfaches Zweiprozessor-System die oben genannte "fail-stop" Eigenschaft erhalten. Koppelt man nun zwei solche Systeme zu einem Vierprozessor-System (*Quadruplexsystem*) zusammen, so kann es wie ein Duplexsystem im obigen Sinn (s. Abb.1.4.1 a,b,c) arbeiten. Ein Beispiel dafür ist das STRATUS-System (s. Abschnitt 3.1).
Soll dagegen das Zweiprozessorsystem fehlertolerant weiterarbeiten, muß zusätzlich zu einer

automatischen Entscheidungsinstanz (*voter*) ein drittes Ergebnis, und damit ein dritter Prozessor, existieren. Diese zusätzliche Redundanz führt uns zum klassischen Zwei-aus-drei-System (Triple Modular Redundancy, TMR), bei dem die übereinstimmenden Ergebnisse der Mehrheit (*Majority voting*) als das richtige Ergebnis von einer Entscheidungseinheit selektiert wird (s. Abb. 1.4.2a).

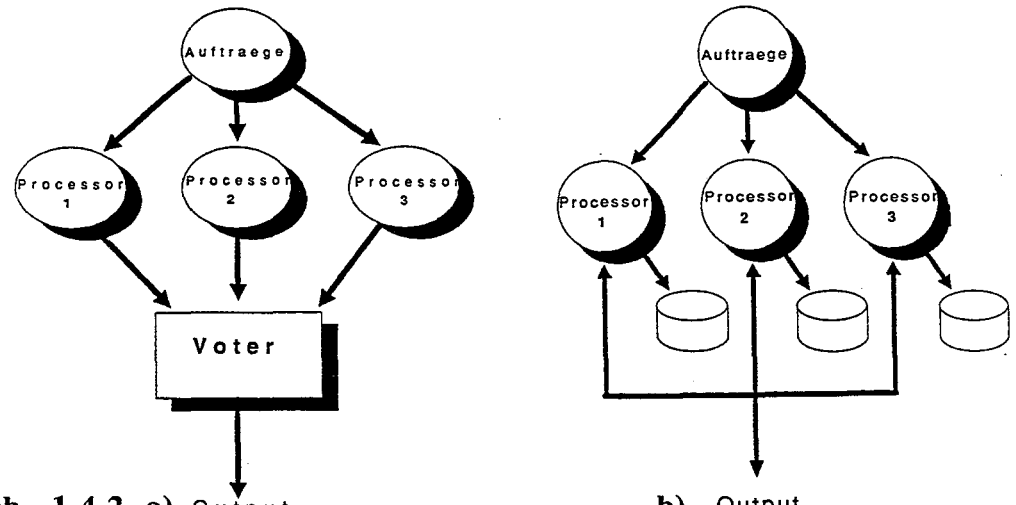


Abb. 1.4.2 a) b)

Mit speziellen Protokollen läßt sich auch die voting-Funktionen verteilt auf allen drei Systemen durchführen, ohne einen zentralen, unüberprüften Voter zu benutzen (s. Abschnitt 3.3).
Sind nicht nur drei, sondern viele (N Stück) Einheiten vorhanden, so spricht man von einem *NMR-System*.

Der Vorteil des Majority-Voting Verfahrens liegt in der sofortigen Verfügbarkeit des richtigen Ergebnisses, so daß nach außen hin der Fehler nicht weitergegeben wird (*statische, maskierende Fehlertoleranz*). Der Nachteil besteht im hohen Hardwareaufwand und der kritischen Implementierung des (ultrazuverlässigen!) Voters.

Bei all diesen Maßnahmen sollte man aber bedenken, daß grundsätzlich derartige Maßnahmen auf Chip-Ebene effektiver sind als auf der Ebene von Systemkomponenten wie zum Beispiel in Abbildung 1.4.2b. Eine Einführung von fehlerkorrigierenden Codes in der CPU, im Speicher, auf den Datenpfaden und in den Peripheriebausteinen ist deshalb nicht nur zur Fehlerkorrektur bzw. Fehlererkennung sinnvoll, sondern führt bereits im Chip zu der günstigen fail-stop Eigenschaft und vermeidet hardwareaufwendige Fehlermaskierung. Andererseits kann die Berechnung und der Vergleich der Endergebnisse (*End-to-end-Strategie*) wie in Abb. 1.4.2b durchaus "normale", nicht-fehlertolerante Hardware benutzen und verhindert die Auswirkungen von unerwarteten und unbekannten Fehlern, die irgendwo im Gesamtsystem auftreten können. Das effektivste Konzept besteht deshalb in der Kombination beider Ansätze zu einem System, das Fehlertoleranz auf allen Ebenen, vom Chip bis zum kompletten System, vorsieht.

Softwareredundanz

Möchte man sich auch gegen mögliche Fehler in der Software schützen, so bietet sich als ein populäres Verfahren das sog. *N-version-programming* (*diversitäre Programmierung*) an. Hierbei werden von verschiedenen Programmierteams, teilweise sogar in verschiedenen Programmiersprachen und Betriebssystemen, verschiedene Programmversionen von ein und derselben Spezifikation erstellt. Ähnlich dem voting aus Abb. 1.4.2a wird nun zur Laufzeit von einer

voting-Einheit das Programmierergebnis der Mehrheit ausgesucht und ausgegeben. Der Vorteil dieser Methode liegt zweifelsohne darin, daß bisher unentdeckte Fehler der Compiler, der Laufzeitbibliotheken und des Betriebssystems auf elegante Weise ausgemerzt werden können (maskierende Fehlertoleranz), ohne dabei einen enormen Aufwand für Tests treiben zu müssen.

Nachteilig wirkt sich allerdings neben dem höheren Programmieraufwand die Tatsache aus, daß auch mit dieser Methode keine Spezifikationsfehler gefunden werden können. Experimentelle Untersuchungen zeigten weiterhin, daß auch verschiedene Teams die gleichen, problemspezifischen Programmierfehler machen können [AV]. Zusätzlich bildet die Funktion des Voters eine Quelle von Problemen, da auch unterschiedlich formatierte, aber inhaltlich gleiche Ergebnisse (z.B. das Ausgabeformat bei Real-Zahlen, die Ausgabe von ASCII-Zeichenketten) als "gleich" erkannt werden müssen.

Im Unterschied zum N-version-programming zeigt sich bei Spezifikationsdiversität ein interessanter Effekt. Wird jede Spezifikation einer anderen Arbeitsgruppe übertragen, so machen die verschiedenen Arbeitsgruppen - im Unterschied zur Programmierung - sehr unterschiedliche Fehler und erlauben so in der Rückkorrektur eine einheitliche, konsistente Spezifikation gemäß den Anforderungen [KEL].

Im Vergleich zu den Fehlertoleranztechniken bei der Hardware fällt auf, daß letztere wesentlich öfter verwendet werden und deshalb besser untersucht und beherrscht sind, obwohl sie durchaus nicht immer wichtiger sind.

1.4.2 Interprozessorkommunikation

Auch für die Übermittlung von Nachrichten haben sich einige Standardverfahren herausgebildet, auftretende Fehler zu behandeln. Betrachten wir dazu das *Schichtenmodell* für die Kommunikation zweier Prozesse auf zwei verschiedenen Rechnern. Nach dem *O*pen *S*ystem *I*nterconnect (*OSI*) Modell der *I*nternationalen *S*tandardisierungs-*O*rganisation (*ISO*) sind die verschiedenen, zu einer Kommunikation zwischen zwei Systemprozessen nötigen Programmteile in Funktionsschichten aufgeteilt. Jede Schicht benutzt nur die Dienste der direkt darunter liegenden Schicht, um ihre Aufgaben zu erfüllen. Dabei werden die Benutzeranforderungen in Absprache mit der gleichen, funktionsentsprechenden Schicht auf den anderen Rechnern fehlertolerant verwirklicht; alle tieferen Schichten sind für die miteinander kommunizierenden Schichten gleicher Funktionalität transparent. Die Kommunikation zwischen den gleichen Schichten zweier Rechner über transparente Zwischenschichten läßt sich als eine *virtuelle Kommunikation* ansehen. In Abbildung 1.4.3 sind die Schichten einer solchen Kommunikation gezeigt.
Dabei sind folgende Funktionen gegeben:

Schicht 1: *Physikal. Übertragung*
Auf der Ebene der Übertragung gibt es viele verschiedenen Standards, beispielsweise Koaxialkabel, Breitbandkabel, verdrillte Zweidrahtleitung, Glasfaser usw. Alle elektronischen und optischen Verstärker und Treiber gehören zu dieser Schicht.

Schicht 2: *Verbindung*
Der Aufbau einer Verbindung zwischen dem Rechner und dem Netzwerk wird durch ein Verfahren (Protokoll) der Zugriffskontrolle geregelt. Diese logische Ebene stellt fest, ob beispielsweise Interfe-

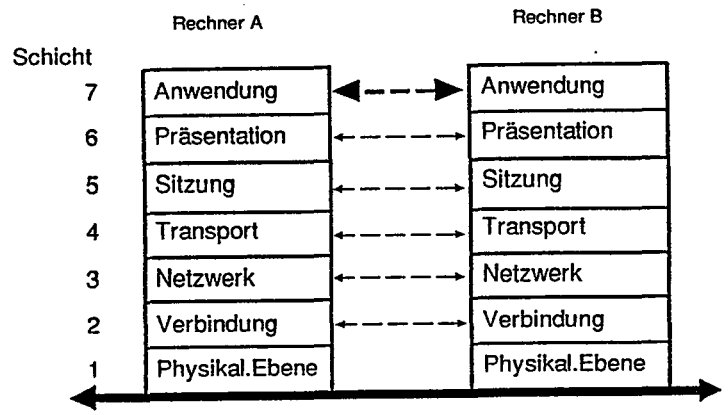


Abb. 1.4.3 Das OSI Schichtenmodell

renzen mit Sendungen anderer Rechner auftreten (z.B. CSMA/CD), ob eine Zugriffsberechtigung vorliegt (z.B. Token Ring oder Bus) und dergleichen mehr. Dabei wird das Format und die Dauer der Übertragung (*Datenframe*) festgelegt (z.B. Ethernet: 22 ByteKopf, Daten, 4 Byte Prüfsumme) sowie feste, einzigartige Adressen für Sender und Empfänger verwendet.
Auch das Auftreten von Fehlern der Physikalischen Ebene wird hier registriert.

Schicht 3: Netzwerk
Von der Netzwerkschicht wird ein einfacher Datagramm-Übermittlungsdienst angeboten, der logische Adressen in physikalische Adresskennungen für die Schicht 2 übersetzt (Beispiel: Internet Protokoll IP), eine Wiederholung eines fehlerhaften Datenframes durchführt und dergleichen mehr.

Schicht 4: Transport
Die Transportschicht stellt auf Wunsch über den Datagramm-Dienst hinaus auch einen gesicherten Datentransport zur Verfügung. Dazu werden beliebig lange Datenpakete in kleinere Normpakete (Datenframes) zertrennt, an Schicht 3 übergeben und beim Empfänger in der richtigen Reihenfolge wieder zusammengesetzt sowie die Fehlermeldungen und Daten von Schicht 3 ausgewertet und danach Maßnahmen wie Sendewiederholung etc. getroffen. Anfangs- und Endpunkte der Kommunikation sind hier definiert (TCP-Protokoll: "Ports").
Um auch unterschiedlichen Einsatzbedingungen gerecht zu werden, wurden von der ISO vier verschiedene Fehlertoleranzanforderungsprofile für die Transportschicht definiert, die von einfacher, völlig ungesicherter Kommunikation bis zu der vollen Ausnutzung aller oben geschilderten Maßnahmen reichen. Das erwähnte TCP-Protokoll implementiert dabei das anspruchsvollste Profil.

Schicht 5: Sitzung
In der Sitzungsschicht wird die Information zur Eröffnung, zum Betrieb und zum Abschluß der Übertragung verwaltet. Vielfach ist diese Schicht ins Betriebssystem eingebunden (Beispiel: BSD-Unix mit "sockets") und benutzt die Kommunikationsendpunkte von Schicht 4.

Schicht 6: Präsentation
In dieser Schicht wird die Kodierung der Daten, je nach Anwendung, durchgeführt (Beispiel: Binäre, ASCII-Kodierung).

Schicht 7: Anwendungen

Diese Schicht ist anwenderspezifisch gestaltet und stellt sehr spezifische Dienste zur Verfügung, je nachdem, ob Normen für einen Datenaustausch einer Büroanwendung (z.B. TOP), einer Fabriksteuerung (z.B. MAP) oder einer anderen Benutzergruppe festgelegt werden müssen.

Jede Schicht fügt zu den Daten der übergeordneten Schicht ihre eigene Kontrollinformation hinzu und erweitert damit die Nachrichtengröße. In der folgenden Abbildung 1.4.4 ist dies für die ersten drei Schichten verdeutlicht.

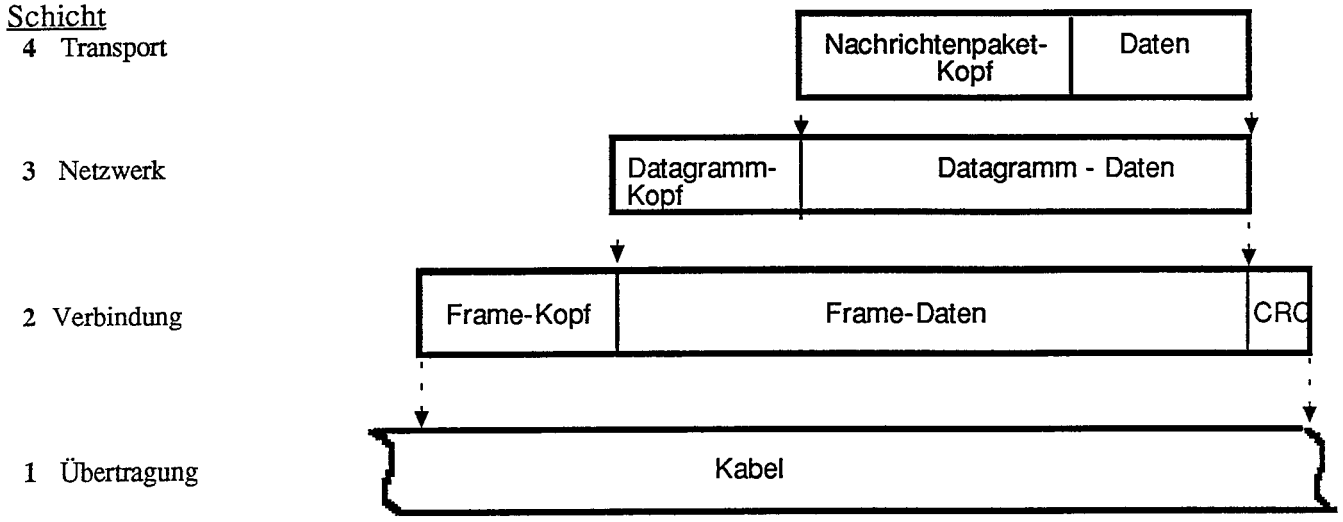


Abb. 1.4.4 Verpackung der Informationen

Die Übertragungsprotokolle der verschiedenen Schichten sind so ausgelegt, daß sie an den von der ISO spezifizierten Funktions-Schnittstellen (virtuelle Kommunikation) leicht getrennt werden können. Damit ist es möglich, Datenübertragung und Fehlerbehandlung separat einem Kommunikationsprozessor zu übergeben, der den Hauptprozessor von dieser Arbeit befreit (s. Abschnitt 1.3.2).

Sind andererseits für bestimmte Kommunikationszwecke (z.B. Implementierung einer Nachrichtenkopplung auf speichergekoppelten Systemen) die Funktionen mancher Schichten unnötig, so läßt sich durch "leere Schichten" auch schnelle Kommunikation verwirklichen, ohne damit die Schnittstelle zum Anwenderprozeß verändern zu müssen. Damit ist es möglich, verteilte Applikationen sowohl auf LANs als auch auf Multiprozessorsystemen laufen zu lassen, je nach vorhandener Konfiguration.

Die Schichtung hat außerdem den Vorteil, daß die Fehlertoleranz für jede Dienstleistung, falls möglich, transparent für den Auftraggeber abgewickelt wird. Beispielsweise werden Nachrichtenwiederholungen, der Ausfall von Netzbrücken und die Auswahl alternativer Kommunikationswege von den unteren Übertragungsschichten verwirklicht, ohne daß der Anwenderprozeß sich darum kümmern muß. Dieses Fehlertoleranzprinzip ist auch für die Validierung eines Systems (s. Abschnitt 3.4.4) von großem Vorteil.

1.5 Test- und Diagnosemodelle

In der Einleitung (Kapitel 1.1) wurde bereits auf die zunehmende Aufgabenvielfalt und Komplexität heutiger Rechnersysteme verwiesen. Die neuen Anwendungen dieser Systeme in der Luft- und Raumfahrt, in der Medizintechnik und in anderen sicherheitskritischen Bereichen wie Verkehrskontrolle und Atomreaktoren verlangen, daß die technischen Systeme - trotz wachsender Komplexität und damit höherer Ausfallwahrscheinlichkeit - ausfallsicherer werden.
Dies führte zu verstärkten Bemühungen in den 70-er Jahren, Entwurfsfehler bereits im Ansatz zu vermeiden durch systematische und genaue Schaltungsentwicklung mit CAD und CAM für die Hardware sowie durch analytische Programmentwicklung mit rechnergestützter Validierung, Verifikation und Dokumentation der Software (*Fehlervermeidende Systeme*).

Andererseits wurden solche Systeme erforscht, entwickelt und gebaut, die beim Betrieb auftretende Fehler rechtzeitig erkennen und diese ohne Eingriffe des Benutzers selbsttätig behandeln (*Fehlertolerante Systeme*).

Da man einerseits durch die unverhältnismäßig hohen Kosten von Folgefehlern zuerst einmal alle Fehler vermeiden muß, die vermeidbar sind, andererseits aber man auch bei gutem Hardwaredesign und guten Programmiersystemen prinzipiell nie Fehler durch ausfallende Bauteile oder durch falsche Programmierung ausschließen kann, sind für zuverlässige Systeme sicher beide Ansätze notwendig. Auf die zweite Möglichkeit soll in dieser Arbeit näher eingegangen werden. Dazu werden in den folgenden beiden Kapiteln 1.5.1 und 1.5.2 zunächst die wichtigsten Modelle zur Fehlerdiagnose vorgestellt.

Tests

Um den so unterschiedlichen, komplizierten Testmethoden mit Oszilloskop, Logikanalysator und Testsoftware gerecht zu werden, beschränken sich die Modellannahmen über die zu der Fehlerdiagnose eingesetzten Tests weitgehend auf die entscheidende Aussage jedes Tests, ob die getestete Elektronik defekt ist oder nicht. Entsprechend wird auch das Gesamtsystem in testbare Einheiten u_i mit statistisch unabhängigen Lebensdauern R_i aufgeteilt, wobei die Einheiten nur zwei Zustände haben können: intakt oder defekt. Je nach Modellannahmen können Einheiten auch für den Test einer anderen Einheit eingesetzt werden.

Dabei sind die Begriffe "Einheit" und "Test" sehr allgemein aufzufassen. Eine Einheit muß nicht nur ein Mikroprozessor sein, der beispielweise mit einem Speichertest die Einheit "RAM" testet, sondern kann auch ein einzelner Logikbaustein oder auch ein kompletter Rechner sein. Betrachten wir beispielsweise das fehlertolerante Multiprozessorsystem ATTEMPTO, das unter Mitwirkung des Verfassers konzipiert und implementiert worden ist (s. Kapitel 2.3 und 3.4). ATTEMPTO besteht aus einer Zahl von unabhängigen Single Board Computern, die miteinander gekoppelt sind. Bei fehlertolerantem Betrieb arbeiten mehrere der Single-Board Computer parallel Kopien des gleichen Programms ab und koordinieren sich bei der Ausgabe. Mit dem Begriff "Einheit" kann in diesem System jeder Single Board Computer bezeichnet werden und mit dem Begriff "Test" eine Operation, bei der die Ausgabedaten der verschiedenen Programmkopien auf Übereinstimmung verglichen werden.

Selbsttests

Eine andere, sehr beliebte Form der Fehlererkennung sind *Selbsttests*. Hierbei wird meist der

Prozessor als intakt angenommen, der bei der Initialisierung (Power up, Hardware Reset etc) alle anderen Einheiten des Computers (RAM,ROM, I/O, etc) testet. Bedingt durch die hohe Zahl an Gatterfunktionen treten auch meistens die Fehler im Speicher auf, so daß diese Testform zu befriedigenden Ergebnissen in der Praxis führt.

Allerdings haben moderne Mikroprozessoren auch eine beachtliche Komplexität und damit eine durchaus nichtverschwindende Ausfallwahrscheinlichkeit. Dem versucht man, durch *Prozessorselbsttests* Rechnung zu tragen. Dabei werden bestimmte Instruktionen des Prozessors als intakt vorausgesetzt (*Hardcore*), mit deren Hilfe sich die anderen Maschinenbefehle testen lassen. Leider aber setzen diese Instruktionen bereits so viele Teile vom Prozessor als intakt voraus (z.B. 60% der Chipfläche mit nur 3 Instruktionen [MAE1]), daß der überwiegende Teil der Prozessorfehler zu unvorhersehbaren Ergebnissen führt.
Es ist deshalb sinnvoller, in der Testkonfiguration auch den Defekt eines Prozessors zu berücksichtigen.

1.5.1 Graphentheoretische Ansätze

Betrachten wir nun ein System, bei dem die N Einheiten u_1, ..., u_N sich gegenseitig testen können. Dieser Sachverhalt läßt sich durch einen gerichteten Graphen G(V,E) mit der Knotenmenge V und der Kantenmenge E beschreiben. Die Knoten werden dabei den Einheiten u_i und die Kanten den Testverbindungen im System zugeordnet. Gemäß unseren obigen Überlegungen kann das Testergebnis genau zwei Werte annehmen:

$$t_{ij} = \begin{cases} 0 & u_i \text{ entdeckte keinen Fehler bei } u_j \\ 1 & u_i \text{ fand einen Fehler bei } u_j \end{cases}$$

Der gerichtete Graph wird *Testgraph* genannt; die Testergebnisse sind somit Gewichtungen der Kanten. Ein Beispiel ist in Abb. 1.5.1 zu sehen.
Die geordnete Menge *S* der |E| Testergebnisse wird als *Syndrom* bezeichnet; die der defekten Einheiten als *Fehlerklasse F*. Dabei heißt eine Fehlerklasse F_k *konsistent* zu einem Syndrom S_l, wenn beim Auftreten von F_k das Syndrom S_l beobachtet werden kann.

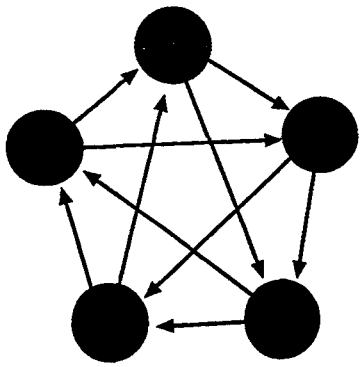


Abb. 1.5.1 Ein Testgraph mit N=5 Einheiten

Das Grundproblem der Diagnose besteht darin, bei gegebenem Testgraphen und gegebenem Syndrom herauszufinden, welche Einheiten defekt sind. Dies läßt sich in folgende Unterprobleme aufgliedern:

* Finde heraus, unter welchen Umständen das Problem lösbar ist.

* Finde *optimale* Testgraphen, die mit möglichst wenigen Tests möglichst viele Fehler lokalisieren.

* Finde Algorithmen für die Diagnose

Das Grundmodell

Als erste gingen Preparata, Metze und Chien in ihrer klassischen Arbeit [PRE] die zwei erstgenannten Probleme an. Sie bezeichneten ein System als *1-Schritt t-Fehler diagnostizierbar* oder auch *t-diagnostizierbar ohne Reparatur* , wenn mit der Kenntnis des Testgraphen und des Syndroms alle fehlerhaften Einheiten sofort lokalisiert werden können, vorausgesetzt, es sind nicht mehr als t Einheiten defekt.

Dagegen wird ein System *sequentiell t-Fehler diagnostizierbar* oder auch *t-diagnostizierbar mit Reparatur* genannt, wenn von maximal t fehlerhaften Einheiten des defekten Zustands mindestens eine Einheit identifiziert werden kann.

Die beiden Annahmen

- Wenn die testende Einheit intakt ist, so ist das Testergebnis korrekt, d.h die Tests sind perfekt (Vollständigkeitsaxiom der Tests) (1.5.1)

- Wenn die testende Einheit defekt ist, so ist das Testergebnis unabhängig vom Zustand der getesteten Einheit und kann beide Werte, 0 oder 1, annehmen. (1.5.2a)

führten zu den notwendigen Bedingungen

- $N \geq 2t + 1$ (1.5.3)
über die Zahl der ausgefallenen Einheiten bei einer korrekten Diagnose von 1-Schrittt-Fehler diagnostizierbare Systeme. Gilt andererseits $N \geq 2t+1$, so läßt sich immer ein Testgraph angeben, der 1-Schritt t-Fehler diagnostizierbar ist.

- Jede Einheit muß mindestens von t anderen Einheiten getestet werden. (1.5.4)

Mit diesen beiden Bedingungen läßt sich eine Klasse D_{1t} von *optimalen Testgraphen* definieren, für die $N=2t+1$ gilt und jede Einheit von genau t anderen Einheiten getestet wird.

Zu den obigen beiden notwendigen Bedingungen fanden Hakimi und Amin [HAK] auch hinreichende Bedingungen für 1-Schritt t-Fehler Diagnostizierbarkeit. Eine davon macht eine Aussage über den Zusammenhang K(G) des Graphen, d.h. über die kleinste Zahl von Knoten, deren Entnahme den Graphen nicht mehr streng zusammenhängen läßt:

- Ist $N \geq 2t + 1$ und $K(G) \geq t$, so ist G t-Fehler diagnostizierbar ohne Reparatur.

Beispielsweise ist ein n-dim. Bool'scher Würfel mit $n \geq 3$ n-Fehler diagnostizierbar ohne Reparatur [ARM]. Weitere Überlegungen zu diesem Modell sind beispielsweise in [MAN], [CIO] oder [SAH] zu finden; ein Diagnosealgorithmus der Laufzeit-Komplexität $O(N^3)$ ist in [MAD] angegeben.

Ein modifiziertes Grundmodell

Eine der ersten, die das Grundmodell der Diagnose von Preparata, Metze und Chien modifizierten und weiterentwickelten, waren Barsi, Grandoni und Maestrini in [BAR1]. Mit der Überlegung, daß die Wahrscheinlichkeit für defekte Systeme relativ gering ist, Tests an anderen defekten Systemen mit dem Ergebnis "korrekt" abzuschließen, änderten sie die Annahme (1.5.2a) in

- Wenn beide Einheiten defekt sind, lautet das Testergebnis richtigerweise immer "defekt". (1.5.2b)

Als Folgerungen der beiden Annahmen (1.5.1) und (1.5.2b) ergeben sich wieder die Bedingungen (1.5.3) und eine weniger restriktive Version von (1.5.4). Damit läßt sich eine Klasse von optimalen Testgraphen definieren, bei der jede Einheit gerade von t anderen Einheiten getestet wird. Infolgedessen enthält die Klasse der optimalen Testgraphen im Sinne von Barsi et alii alle D_{1t}-Testgraphen von Preparata (N=2t+1) sowie alle Testgraphen mit dem D_{1t} Verbindungsschema und geradem N sowie $N-2 > t \geq N/2$.
Ein Algorithmus der Komplexität $O(N^3)$, der maximales t eines Testgraphen bestimmt sowie ein Diagnosealgorithmus der Komplexität $O(N^2)$ ist in [AM4] zu finden.

Das Vergleichstestmodell

Gegenüber dem Grundmodell, das in (1.5.1) nur über die Testergebnisse von intakten Einheit Aussagen macht, nahm Malek in [MAL] an, daß,

- wenn die testende Einheit defekt ist, das Testergebnis immer "defekt" lautet. (1.5.2c)

Daraus folgt direkt (siehe Tabelle 1.5) $t_{ij} = t_{ji}$

Mit den Modellannahmen muß also keine Testrichtung angegeben werden, da das Ergebnis des symmetrischen Tests sich automatisch ergibt. Der Testgraph ist somit ungerichtet; ein Test läßt sich als *Vergleich* der Zustände beider Einheiten deuten und wird deshalb *Vergleichstest* genannt. Da Malek keine notwendigen und hinreichenden Bedingungen der t-Diagnostizierbarkeit (ohne Reparatur) angab, wurden diese von Mitgliedern unserer Arbeitsgruppe entwickelt. In [AM1] und [AM5] ist dieses näher ausgeführt.

Optimale Testgraphen, also Testgraphen, die bei gegebener maximalen Zahl t defekter Einheiten möglichst wenig Knoten und Kanten enthalten, werden für dieses Diagnosemodell dadurch konstruiert, daß vom vollständig vernetzten Graphen mit N=t+2 Knoten eine Kante gestrichen wird. Dabei werden mindestens t Vergleiche (Kanten) pro Knoten und mindestens $\lceil N\ t/2 \rceil$ Vergleiche im Gesamtsystem angestellt. Zur Illustration seien einige Beispiele solcher t-optimaler Testgraphen in der Abbildung 1.5.2 gezeigt.

Betrachtet man zusätzlich nur noch diejenigen optimalen Testgraphen, bei denen die intakten Einheiten durch t_{ij} = "intakt" direkt gefunden werden können, so werden diese Testgraphen als *streng t-optimale* Testgraphen bezeichnet. Für diese Klasse von Testgraphen lassen sich relativ einfache Dignosealgorithmen konstruieren; in [AM1, AM4] ist ein sequentieller, zentral durchgeführter Diagnosealgorithmus der Ordnung $O(N^2)$ sowie ein paralleler, auf einer Baummaschine ausführbarer Algorithmus der Ordnung O(N-1+ld(N)) angegeben.
Die streng t-optimalen Testgraphen haben dabei gegenüber den einfachen t-optimalen Testgraphen den Vorteil, nur genau $\lceil N\ t/2 \rceil$ Vergleiche (Tests) für eine schnelle Diagnose zu benötigen. Dabei

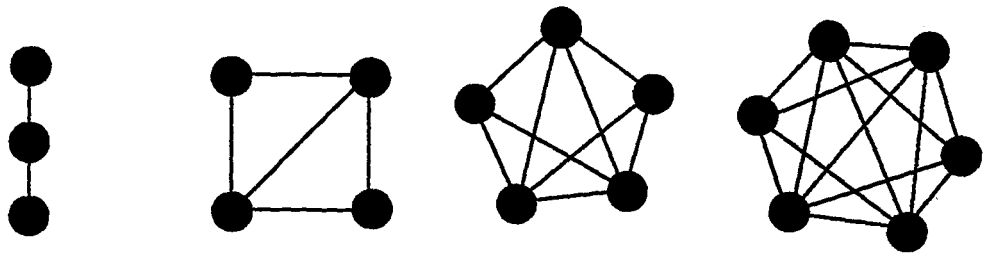


Abb. 1.5.2 Beispiele t-optimaler Testgraphen

muß allerdings mindestens eine zusätzliche Einheit, also insgesamt mindestens N=t+3 Einheiten, vorhanden sein. Die Diagnosealgorithmen für t-diagnostizierbare Testgraphen ergeben sich durch zusätzliche Schritte zu den Algorithmen der streng t-diagnostizierbaren Testgraphen.

1.5.2 Ansätze der Statistik und Mustererkennung

Das von Preparata et alii entwickelte Diagnosemodell ist in bestimmten Situationen nicht anwendbar, ebensowenig wie die anderen graphentheoretischen Modelle des vorigen Abschnitts. Der Grund dafür liegt in der Annahme der Modelle, daß beim Einsatz von intakten Einheiten immer korrekte Testergebnisse erzielt werden (*Vollständigkeitsannahme* der Tests). Leider ist diese Annahme aber nicht immer zutreffend. Bei hochkomplexen Bauelementen können die Tests meist nur grob die wichtigsten Funktionen testen, da ein systematisches Überführen des Bauelements in alle möglichen Zustände wegen der Komplexität der Bauelemente in einer vertretbaren Testzeit nicht möglich ist. Das Testergebnis kann somit nur noch eine Wahrscheinlichkeitsaussage über den Zustand des Bauelements darstellen, selbst wenn die testende Einheit intakt ist.

Anstelle der graphentheoretischen Modelle werden stattdessen Diagnoseverfahren benötigt, bei welchen von vorliegenden Testergebnissen mit bestimmter Wahrscheinlichkeit auf den Systemzustand geschlossen werden kann.

Probabilistische Diagnose

In einem ersten Schritt kann man die Ausfallwahrscheinlichkeiten der Einheiten für die Diagnose eines unentscheidbaren Syndroms zu Hilfe nehmen. Bezeichnen wir eine Menge von defekten Einheiten als Fehlerklasse F_i, so kann man als wahrscheinlichste Fehlerklasse des Systems diejenige Fehlerklasse F ansehen, die mit dem beobachteten Syndrom S konsistent ist (entsprechend der Annahmen (1.5.1) und (1.5.2a) von Preparata o.ä.) und die größte Auftrittswahrscheinlichkeit hat

$$P(F) := \max_i P(F_i) \qquad F_i \text{ konsistent zu } S$$

Im allgemeinen kann es mehrere F_i geben, die obige Bedingung erfüllen und gleiches $P(F_i)$ haben. Obengenannte Diagnosevorschrift arbeiteten Maheshwari und Hakimi in [MAH] genauer aus und leiteten Bedingungen für eine Diagnose ab; Fujiwara und Kinoshita gaben in [FUJI] die Konstruktion der dazu optimalen Testgraphen an.

In einem zweiten Schritt lassen sich nun zusätzlich die Wahrscheinlichkeiten dafür in die Diagnose einbeziehen, daß keine vollständigen Tests existieren oder die Tests (z.B. durch transiente Fehler) gestört sein können. Mit der Kenntnis dieser Parameter läßt sich die Wahrscheinlichkeit einer richtigen Diagnose erhöhen, wie dies beispielsweise von Blount [BLO] vorgeschlagen wurde. Die Diagnoseannahmen von Preparata, Barsi und Malek lassen sich als Spezialfälle dieses sehr allgemeinen Ansatzes in der folgenden Tabelle gegenüberstellen.

Zustand der Einheiten	*Testergebnis*	$P(t_{ij}\|(u_i u_j))$ *pro Modell*			
0=intakt,1=defekt (u_i u_j)	t_{ij}	Blount	Preparata	Barsi	Malek
0 0	0	p_{ij}	1	1	1
	1	$1-p_{ij}$	0	0	0
0 1	0	$1-r_{ij}$	0	0	0
	1	r_{ij}	1	1	1
1 0	0	q_{ij}	p	p	0
	1	$1-q_{ij}$	1-p	1-p	1
1 1	0	$1-s_{ij}$	p	0	0
	1	s_{ij}	1-p	1	1

Tabelle 1.5 Gegenüberstellung der Modellannahmen

Welches ist nun die günstigste Strategie (Algorithmus), um mit der Kenntnis der Parameter p,q,r und s eine korrekte Diagnose in einem System zu erzielen?

Sei die Wahrscheinlichkeit, daß alle im System auftretenden Defekte F_k bei gegebener Diagnosevorschrift σ auch richtig diagnostiziert werden, als *Diagnostizierbarkeit* D_{sys} bezeichnet (s. [DAL1],S.62). Es gilt

$$D_{sys} := \sum_k P(F_k \text{ erkannt} \mid F_k)\, P(F_k) \qquad (1.5.5)$$

wobei jedes F_k mit Hilfe von σ diagnostiziert wird: $F_k = \sigma(S_l)$. Die Wahrscheinlichkeit, die Fehlerklasse richtig zu erkennen, ist dabei

$$P(F_k \text{ erkannt} \mid F_k) = \sum_l P(S_l \mid F_k)$$

Damit ergibt sich

$$D_{sys} := \sum_k \sum_l P(S_l \mid F_k)\, P(F_k) \qquad \text{mit } F_k = \sigma(S_l) \qquad (1.5.6)$$

Eine optimale Diagnosevorschrift maximiert D_{sys} oder

$$P(S_l, \sigma_{opt}(S_l)) = \max_k P(S_l \mid F_k)\, P(F_k) \qquad (1.5.7)$$

Hierbei haben wir nicht nur die günstigste Strategie der probabilistischen Diagnose, sondern auch einen Diagnosealgorithmus dazu gefunden: Wähle die Diagnosevorschrift σ_{opt} bzw. die Fehlerklasse F_i, für die ein Syndrom S_l so, daß die bedingte Syndromauftrittswahrscheinlichkeit maximiert wird.
Der Vorteil dieser Diagnoseart gegenüber den deterministischen Diagnosearten besteht in der verbesserten Diagnostizierbarkeit gerade bei unzuverlässigen, ungleichen Einheiten mit ungleichen

Ausfallraten (s.[BR5],pp.47), wie dies bei unterschiedlich komplexen Chips und Platinenkomponenten in der Praxis oft auftritt.

Probabilistische Diagnose mit Kosten

Sind mit verschiedenen Diagnosen auch zusätzlich Kosten verbunden, wie beispielsweise die Reparaturkosten einer Einheit, oder die Kosten, bei falscher Reparatur durch zusätzlich nötige Tests den Systemausfall zu verlängern, so werden die Diagnoseentscheidungen noch mit einem Kostenfaktor L_{kl} gewichtet. Das neue Diagnoseziel besteht darin, die erwarteten Gesamtkosten (*das Risiko*) R_σ der Diagnose zu minimieren

$$R_\sigma := \sum_k \sum_l L_{k\sigma} P(S_l \mid F_k)\, P(F_k) \qquad \text{mit } F_k = \sigma(S_l) \qquad (1.5.8)$$

Wie nicht anders zu erwarten, verursacht die kostengünstigste, probabilistische Diagnose immer gleiche oder geringere Kosten als die deterministische oder rein probabilistische Diagnose [BR5],pp.58 .

Ein Nachteil der probabilistische Diagnose ist der Verlust der Möglichkeit, durch die graphentheoretischen Modellbezüge mittels konstruktiv-gezielt verbesserten Testgraphen auch eine verbesserte Diagnostizierbarkeit zu gewinnen. Dies wirkt sich allerdings nur bei Systemen aus, deren Kommunikationsstruktur bei der Konstruktion oder beim Testen frei gewählt werden kann. Ist dies durch die Anwendung von vornherein beschränkt, so sind die graphentheoretischen Modelle in diesem Fall nur eingeschränkt brauchbar.

Ein anderer Nachteil, der gegen die Anwendung der probabilistische Diagnose in der Praxis spricht, ist die Unkenntnis der bei der Diagnose verwendeten Parameter. Lassen sich die Ausfallwahrscheinlichkeiten der Chips meist noch gut mit Hilfe von Datenhandbüchern (*Military Standard*) schätzen, so ist die Ausfallrate des betrachteten Gesamtsystems aus Chips, Karten, Steckern, Lüftern und anderer mechanisch-elektrischen Komponenten wesentlich schwieriger zu bestimmen. Betrachten wir zusätzlich noch die für die Diagnose eingesetzten Tests, so sind wir bezüglich ihrer Abdeckung und Diagnosefähigkeit vollständig auf Spekulationen angewiesen, da weder eine Verifikation der Tests (Geheimhaltung der detaillierten inneren Chipstruktur und -aufbau durch viele Firmen) noch eine experimentelle Überprüfung (unbekannte Zahl möglicher Fehler) durchführbar ist.

Ein Ansatz zur Überwindung dieser Probleme besteht in der sukzessiven Schätzung der verwendeten Parameter. Dabei sollen bei jeder neuen Schätzung die vorherigen Diagnose- und Reparaturerfahrungen eingehen. Diese Problematik läßt sich mit den Methoden des iterativen Erlernens von Klassifizierungsvorschriften behandeln, wie sie in der klassischen Mustererkennungstheorie üblich sind (vgl. Kapitel 4.1). Hierbei wird das Risiko in Gleichung (1.5.8) minimiert, indem die Parameter nach jeder Reparatur neu mit dem kleinsten quadratischen Fehler stochastisch approximiert werden.

Eine genaue Analyse dieses Ansatzes in [BR3] zeigt aber, daß diese Anwendung der stochastischen Approximation nicht unproblematisch ist. Da wir von vornherein die Diagnose (und damit die Reparatur) auf geschätzten Parametern aufbauen, ist keine objektive Beobachtung der Ausfälle und der Ausfallrate möglich. Der Übergang des Systems von einem defekten Zustand in einen Zustand mit weniger defekten Einheiten ist mit der probabilistischen Diagnose nur mit einer bestimmten Wahrscheinlichkeit gegeben, die von dem Parametervektor des Zustandes abhängt und nicht von

früheren Reparaturversuchen.

Die Zustandsübergänge im System lassen sich folglich als Markovkette beschreiben, deren Komplexität durch die Größe des Zustandsraums bei vielen Einheiten zu groß für eine effektive Berechnung wird. Beschränken wir die Betrachtungen auf symmetrische Strukturen, so zeigt sich als Ergebnis, daß beispielsweise die Ausfallraten λ_i der Einheiten sehr genau gelernt werden können. In den meisten Fällen ist die Diagnose richtig, so daß die falschen Entscheidungen (und Reparaturen) nur als stochastische Abweichungen auftreten. Selbst bei einer vorsichtigen, pessimistischen Diagnosestrategie, die bei unklaren bzw. unwahrscheinlichen Syndromen gleich das ganze System ersetzt, ist die dadurch einseitig verschobene, gelernte Ausfallrate nur um ca. 12% gegenüber der tatsächlichen erhöht.

Sequentielle Diagnose und Reparatur

Die bisher beschriebenen Diagnoseverfahren diagnostizieren ein System von N Einheiten in einem Schritt. In Systemen mit sehr vielen Einheiten ist dieses Vorgehen aber durch zwei Tatsachen problematisch:

- Auch bei Systemen mit sehr vielen Einheiten gibt es meist nur eine feste, maximale Anzahl von Kommunikations- (und damit Test-) Verbindungen. In allen Diagnosemodellen des vorigen Abschnitts waren aber für die Diagnose von t defekten Einheiten mindestens t direkte Testverbindungen vorausgesetzt. Die Diagnose bleibt damit, unabhängig von der Größe des vernetzten Systems, auf wenige Einheiten beschränkt.

- Auch wenn die Testergebnisse in kodierter Form über Kommunikationsumwege jede Einheit erreichen könnten, verbietet sich die Diagnose großer Systeme durch die Komplexität des Algorithmus. Beispielsweise hat die Diagnose mit dem kleinsten Risiko aus Gleichung (1.5.8) bei N Einheiten und M Tests die Zeitkomplexität der Ordnung $O(2^M 2^{2N}(2N+M))$, s.[BR5], pp.155. Dauert eine Operation 10^{-10}s = 0.1 Nanosekunden, so dauert die Diagnose bei N=1 insgesamt 2^{11} Operationen oder 2 Sekunden; bei N=2 bereits 17 Minuten, bei N=3 rund 4.5 Tage und bei N=4 schon 5 Jahre.

Aus beiden obigen Argumenten kann man folgern, daß es wesentlich besser als eine globale Diagnose für das System einzuleiten ist, Diagnosen nur lokal für wenige Einheiten aber parallel im ganzen System durchzuführen. Wird die Diagnose durch die Entdeckung eines Fehlers, beispielsweise bei periodischen Tests oder Überprüfung der Nachrichten auf Formattreue, Konsistenz und Plausibilität ausgelöst, so kann man beispielsweise durch Ersetzen der als "defekt" erkannten Einheiten die Wahrscheinlichkeit einer korrekten Diagnose für die restlichen Einheiten vergrößern. Möchte man einerseits möglichst wenige Einheiten ersetzen und andererseits aber auch einen möglichst kleinen Bereich des Gesamtnetzes mit Testen und Diagnostizieren belasten, so bietet sich das kostengewichtete Diagnoseverfahren aus Gleichung (1.5.8) an. In [BR5], pp.91 wird gezeigt, wie mit einer Ausweitung des lokalen Diagnosebereichs durch schrittweises Vergrößern des Testgraphen (*Teststufen*) in Form einer *Testaktivierung* der benachbarten Einheiten ein Minimum aus den Kosten einer Fehldiagnose und den Testkosten erreicht wird.

Mit dieser Methode lassen sich auch größere Prozessorfelder diagnostizieren und reparieren. Nehmen wir beispielsweise an, daß die Verbindungsstruktur solcher Prozessorfelder regelmäßig ist und ohne Leitungskreuzungen auf eine Fläche projektierbar ist (Wafer-scale Integration!), so existieren nur drei Grundtypen von Prozessornetzen [BR5, S.82], wie in Abbildung 1.5.3 gezeigt.

Ein Vorschlag für die Architektur eines sochen Prozessorknotens mit seinen Charakteristika ist in [TROB1] zu finden.

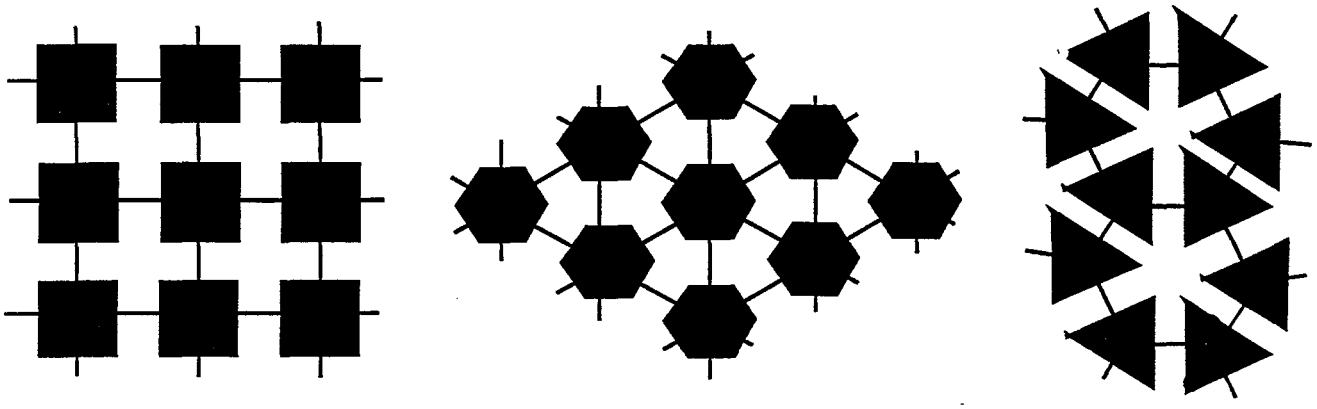


Abb. 1.5.3 Reguläre Flächennetze

Es läßt sich zeigen, daß solche Flächennetze "t-Fehler selbstreparierend" sind, wenn maximal $t \leq 2N^{1/2} - 3$ Einheiten defekt sind [BR5,pp.116].
In diesem Fall können die intakten Nachbarn zuerst alle ihre defekten Nachbarn reparieren, dann die Nachbarn der Nachbarn, und so weiter, bis zum Schluß das gesamte Rechnernetz fehlerfrei ist.
Obwohl auch andere Testalgorithmen für Prozessornetze bei Wafer-Scale Integration entwickelt wurden [TROB2], ist die Anwendung dieser Überlegungen in naher Zukunft allerdings noch nicht abzusehen.

2.0 Parallelarbeit in Multi-Mikroprozessor-Systemen

In den Abschnitten 1.2 und 1.3 sahen wir, daß die Multiprozessorsysteme sich grob in zwei Klassen einteilen lassen: die Prozessorsysteme mit gemeinsamem Speicher (fest- oder speichergekoppelte Systeme) und die Systeme ohne gemeinsamen Speicher (lose oder nachrichtengekoppelte Systeme). In den folgenden Abschnitten soll zur näheren Verdeutlichung der mit der Architektur zusammenhängenden Hard- und Softwareprobleme zuerst das Spektrum der speichergekoppelten *shared memory* Systeme und dann das der nachrichtengekoppelten Systeme anhand einiger Implementierungsbeispiele aus der Literatur kurz dargestellt werden. Dabei wird für die wichtigsten Architekturen aus Abschnitt 1.2 je ein Beispiel einer Implementierung mit ihren signifikantesten Daten kurz vorgestellt.

Aus dem gesamten Spektrum wird im dritten Abschnitt ein bestimmtes System, das fehlertolerante ATTEMPTO-System, ausgewählt und für dieses System konkret die Lösungskonzepte und Implementierungen der vorher diskutierten Probleme geschildert. Dabei werden in diesem Abschnitt fast ausschließlich diejenigen Aspekte von ATTEMPTO besprochen, die für die Problematik der Systemprogrammierung und Koordination von Multiprozessorsystemen ausschlaggebend sind. Die Fehlertoleranzaspekte sind, soweit möglich, davon getrennt und werden erst im nachfolgenden Abschnitt 3 im Kontext mit anderen fehlertoleranten Systemen dargestellt.

2.1 Speichergekoppelte Systeme

Bei den speichergekoppelten Systemen werden im nächsten Abschnitt je ein System mit dem Verbindungsaufbau einer vollständigen Vernetzung (PLURIBUS), eines Kreuzschienenverteilers (C.mmp) und eines mehrstufigen Omega-Verbindungsnetzwerks (NYU) vorgestellt. Dabei werden jeweils die grundsätzliche Hardwarearchitektur sowie wichtige Aspekte der parallelen Programmierung und Fehlertoleranz beschrieben.
Beginnen wir mit einem Beispiel der vollständigen Vernetzung: dem PLURIBUS System.

Pluribus
Das Multiprozessorsystem Pluribus wurde 1972 bei Bolt Beranek&Newman als Nachrichtenverarbeitungssystem für ARPANET konzipiert. Im Ansatz wurden sowohl der Datendurchsatz als auch bereits Fehlertoleranz berücksichtigt [ORNS].

Hardwarearchitektur
Bei der Hardwarearchitektur wurden alle Prozessor/Speicher Paare (CPU+4KB RAM) mit den Ein/Ausgabe- und den Speichereinheiten (2 mal 8kB) über dedizierte Verbindungen gekoppelt, wobei jede Einheit mehrere über einen Bus angeschlossene Module enthielt. In Abbildung 2.1.1 ist ein Überblick gegeben.

Jede I/O Einheit enthält außer den Modulen für die Signalleitungen der zu verarbeitenden Nachrichten und einem Netzteil ein Pseudo-Interrupt Device (PID), das eine hardwareunterstützte, prioritätsgeordnete Warteschlange realisierte, die für den parallelen Programmablauf benötigt wurde.

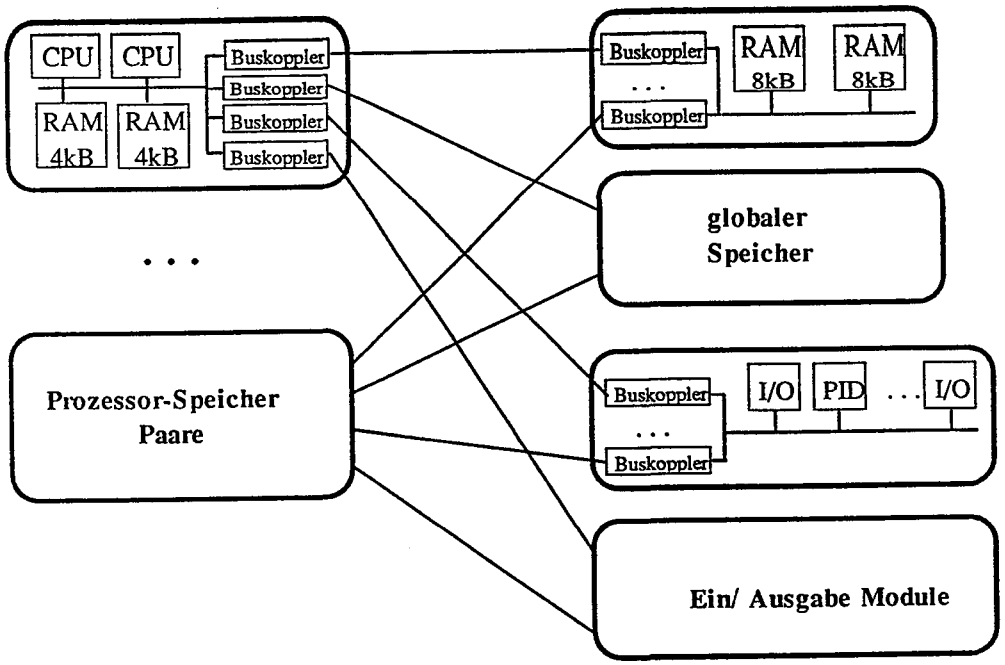


Abb. 2.1.1 Die Pluribus-Architektur

Jeder Buskoppler sowie die Prozessoren können im Fehlerfall abgeschaltet werden, was auch von außen über ein "Network Control Center" möglich ist.

Parallele Programmierung

Das für den Anwendungszweck des Pluribus konzipierte Anwendungsprogramm konnte in viele kleine (max 400ms) parallel ablaufbare Programmstücke ("stripes") zerlegt werden. Jedes dieser mit einer Integer-Zahl (Prozeß-Id) sowie einer Priorität gekennzeichneten Programmstücke wurde als Kopie in jedem lokalen Speicher eines Prozessors gehalten; die Daten dagegen im globalen Speicher.

Erreichte nun eine Nachricht auf einer Signalleitung ein I/O-Modul, so wurde für diese Interruptquelle eine Integer-Zahl im PID abgelegt. Jeder Prozessor durchsuchte nach dem Beenden eines stripes erneut das PID, entnahm die Integer-Zahl mit der höchsten Priorität und arbeitete den entsprechenden stripe ab. Die stripe-Zahlen können aber auch von Prozessoren in das PID plaziert werden, so daß in diesem System mehrere, zentral gehaltene aber dezentral bediente Prozeß-Warteschlangen existieren. Da es sich bei den stripes um zyklische Prozesse handelt, muß nach dem Abarbeiten kein Prozeß-Kontext gespeichert werden und der Prozeß kann jederzeit abgebrochen und mit neuen Daten neu aufgesetzt werden.

Die Prozesse benötigen keinen Austausch von Daten, da die einzelnen Anwenderprozesse vollkommen separat auf den Daten arbeiten (*farming*, s. Abschnitt 4.4.1). Da dabei auch keine Interprozessor-Nachrichten nötig sind, die in Pluribus über Mailboxen des globalen Speichers abgewickelt werden können, ist der Zuwachs an Prozessorleistung bei steigender Prozessorzahl ziemlich linear [THOM].

Fehlertoleranz

Entscheidend für die Möglichkeiten, Fehlertoleranz in diesem System zu realisieren, ist in Pluribus die Tatsache, daß alle Tasks zyklisch wiederholt werden. Ein Abbruch ist, wie gesagt, unproblematisch und führt zu einer Wiederholung der Bearbeitung, die durch ein erneutes Senden

der nicht-quittierten Nachrichten durch die Protokollschichten des ARPANET unterstützt wird. Dadurch entfallen in diesem Real-Time System (vgl. Abschnitt 3.3) die Probleme der Datensicherung, die in den nichtzyklischen Systemen nötig sind.

Die Fehlertoleranzmechanismen beruhen im Wesentlichen auf einer Fehlererkennung durch Datenkonsistenzprüfungen und einer extensiven, überall im System verwendeten Zeitüberwachung mittels watch-dog-timer. So werden beispielsweise die Busaktivität, die Zyklenzeit der Prozesse und die Zugriffshäufigkeiten auf I/O Kanäle überwacht.

Das Betriebssystem "Stage" ist sehr klein und dient der Konsistenz der Zustandstafeln der Resourcen, die von den Prozessoren in einem Votierungsverfahren im globalen Speicher ermittelt werden. Die Zustandstafeln werden von verschiedenen Betriebssystemprozessen ("stages") ermittelt, die bootstrap-ähnlich stufenweise aufeinander aufbauend von jedem Prozessor periodisch abgearbeitet werden und außer dem Auffinden und Initialisieren von Resourcen, wie Speicher, I/O PID und Timer auch Tests der Prüfsummen der Programmteile und der Speicher beinhalten. Außerdem enthalten alle stripes noch einen Selbsttest, so daß über die stripe-Ausführungszeit auch die Selbsttests zeitüberwacht werden. Treten Fehler im Betrieb auf, so werden die Zustandstafeln geändert und die Prozessoren gehen auf eine frühere Betriebsstufe zurück.

Die eingebauten Fehlertoleranzmaßnahmen ermöglichen einen Betrieb von 99,9% der gewünschten Betriebszeit, wobei in Feldstatistiken (3 Maschinen) in 99,76% der Betriebszeit ein Durchsatz von mehr als 92% des Maximalwerts erreicht wurde [ROB].

C.mmp

Eines der ersten Multiprozessor-Projekte war der Carnegie-Mellon multi-mini-processor (C.mmp), der 1971 an der Carnegie-Mellon Universität in USA gestartet wurde.

Hardwarearchitektur

Das Computersystem bestand aus 16 Minicomputern (DEC PDP11/40 mit je 8kB lokalem Speicher und I/O), die über einen Cache und einen Kreuzschienenverteiler (s.Abschnitt 1.2) mit 16 Speichermodulen (2MB) verbunden waren. In der Abbildung 2.1.2 ist ein Überblick über die Architektur gegeben.

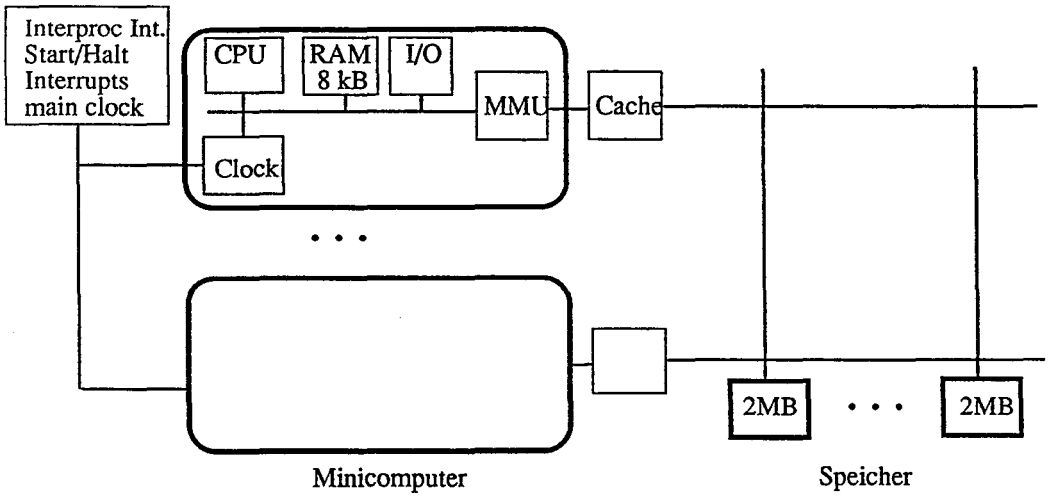


Abb. 2.1.2 Hardwarearchitektur von C.mmp

Parallele Programmierung

Um dem Benutzer eine Kontrolle des Gesamtsystems zu erleichtern, wurde ein Betriebssystem "Hydra" erstellt, dessen Kern allerdings nur die notwendigsten Prozeduren für den gegenseitigen Ausschluß (*mutual exclusion*) beim Zugriff auf globale Datenstrukturen und Prozeduren (*Objekte*) enthielt und alle anderen Beriebssystemdienste als Benutzerprozesse implementierte. War die Datenstruktur gesperrt, so griff der Prozessor nicht iterativ auf die Speicherstelle zu ("spin lock"), sondern wartete auf einen Interrupt, der ihm das Freiwerden anzeigte ("kernel lock"). Da die kritischen Abschnitte kurz (~300µS) waren warteten selbst Prozesse, die 60% ihrer Zeit im Kern verbrachten, weniger als 1% ihrer Zeit auf das Freiwerden der Sperren [JON].

Die in der obigen Abbildung gezeigten Daten-Cache konnten trotz erfolgreicher Fertigstellung nicht benutzt werden, da die damit verbundenen Dateninkonsistenzen (s. Abschnitt 1.3.3) dies unmöglich machten.

Hydra unterstützte auch das moderne Konzept der *capabiltity based objects* sowie eine benutzerdefinierte Scheduling-Politik. Dazu wurde ein Kurzzeit-Scheduler KMPS (Kernel Multi-Programming System) entwickelt, der von benutzerdefinierten Langzeitscheduling-Modulen über Parameter gesteuert wird. Zu den Parametern gehören die für den Prozeß geeigneten Prozessoren sowie die maximalen Speicheranforderungen des Prozesses. Beispiel eines Langzeitschedulers ist ein *Resource Director*, der aus allen in einer Warteschlange befindlichen Prozesse, die durch ihre Prozessor- und I/O Nutzungsanforderungen charakterisiert sind, einen Prozeß aussucht, der die augenblickliche Prozessornutzung geeignet verbessern kann.

Das Hauptproblem dieses für Forschungszwecke geschaffenen Rechners war die geringe Benutzerakzeptanz, was wohl zum einen auf das Fehlen geeigneter Programmierungswerkzeuge, zum anderen aber auf die geringe Fehlertoleranz zurückzuführen war.

Fehlertoleranz

Die Betriebserfahrungen von C.mmp zeigten, daß nicht der Kreuzschienenverteiler die meisten Ausfälle zu verzeichnen hatte, sondern die einzelnen Computereinheiten. Jeder Computer mußte alle 4 Sekunden ein besonderes Bit setzen (*Totmann-Schalter*), um nicht als "defekt" zu gelten und nicht nach einem Hardware-Reset von einem zufällig ausgesuchten Prozessor getestet zu werden.

Obwohl diese Prozedur effektiv gegenüber permanenten Fehlern ist, wurden die Benutzerprogramme auch bei den häufig auftretenden, transienten Fehlern abgebrochen und neu gestartet. Dies war notwendig, da beispielsweise das durch transiente Fehler gestörte Speicher-Busprotokoll im Kreuzschienenverteiler den Speicherweg unbemerkt (und mit ihm alle darauf wartenden Prozessoren) dauerhaft blockieren konnte. Nur ein manueller Hardware-Reset konnte diese Situation bereinigen, wobei alle Benutzerdaten gelöscht wurden. Der Benutzer mußte selbst Sorge dafür tragen, daß wichtige Informationen regelmäßig abgespeichert wurden (s. TANDEM System, Abschnitt 3.2), so daß das System nur geringe Benutzerakzeptanz fand.

Um die Unzufriedenheit der Benutzer durch das häufige Abbrechen der Programme zu verhindern, wurde anstelle der Test-und-Restart Strategie für jede Fehlerklasse und jeden Computer ein "Fehlerzähler" eingerichtet, der über periodisches Shiften die *Fehlerfrequenz* (Zahl der Fehler pro Zeiteinheit) anzeigte. Überschritt der Fehlerzähler einen definierten Schwellwert, so wurde das Teil ausgetauscht. Interessanterweise wurden viele Hardwarefehler nicht durch dedizierte Hardware-

Hardwaretests (Parity etc) gefunden, sondern durch Konsistenzüberprüfungen (Datzentypen etc.), wobei zur genauen Fehlerlokalisierung menschliches Eingreifen nötig war.

NYU

Die Aktivitäten der New York University (NYU) Ultracomputer-Gruppe begannen Anfangs der 80-Jahre und hatten zum Ziel, das Grundproblem der *shared memory* Architektur, die Konflikte beim gleichzeitigen Zugriff von sehr vielen Prozessoren (mehrere Tausend) auf die selbe Speicherstelle (*Flaschenhals*) zu lösen. Dies gelang ihnen in einem bemerkenswerten Ansatz [GOTT2], der deshalb kurz beschrieben werden soll.

Hardwarearchitektur

Zur Verbindung der geplanten 4096-Prozessor Maschine mit den Speichermodulen entschieden sich die Designer für ein Mehrstufen-Verbindungsnetzwerk vom Omega Typ (s. Abschnitt 1.2). In der aktuellen Implementierung verbindet es 8 Prozessoren (MC68010 mit Weitek FPU) mit 8 Speichermodulen (1MB).

Hervorstechendstes Merkmal des Omega-Netzwerks ist die in den Speicherkopplungsknoten eingebaute Fähigkeit, mehrere Datenpakete (Nachrichten) für die selbe Speicherzelle mittel eines schnellen VLSI-Chips miteinander in einer Nachricht zu kombinieren. Dies gilt auch für die *fetch-and add* Primitive aus Abschnitt 1.3.3, die damit jeweils atomar ausgeführt (*serialisiert*) werden. Zwar ist die Reihenfolge dieser Zugriffe nicht vorher festgelegt, aber sie werden ohne Zeitverlust in einem Zyklus seriell ausgeführt.

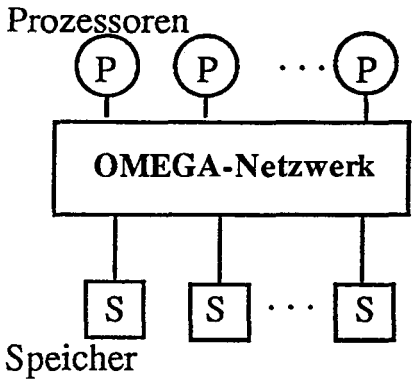


Abb. 2.1.3 Die NYU Ultracomputer Konfiguration

Damit vermeidet dieses Hardwarefeature auch eine Blockierung des Netzwerks durch blockierte Verbindungspunkte (*hot spots*): Bei großem Nachrichtenandrang (Zugriffsoperationen) auf die selbe Speicherzelle werden durch Warteschlangen nicht nur die Nachrichtenübertragung in diesem Verbindungsknoten, sondern auch rückwirkend in allen verbindungsmäßig davor gelagerten (und damit fast im gesamten Netzwerk) blockiert. In Abbildung 2.1.4 ist links das Blockschema eines solchen Elements und rechts der simulierte Durchsatz eines 64x64 Omega Netzwerks mit und ohne Nachrichtenkombinierung gezeigt (aus [ALM],S.297ff).

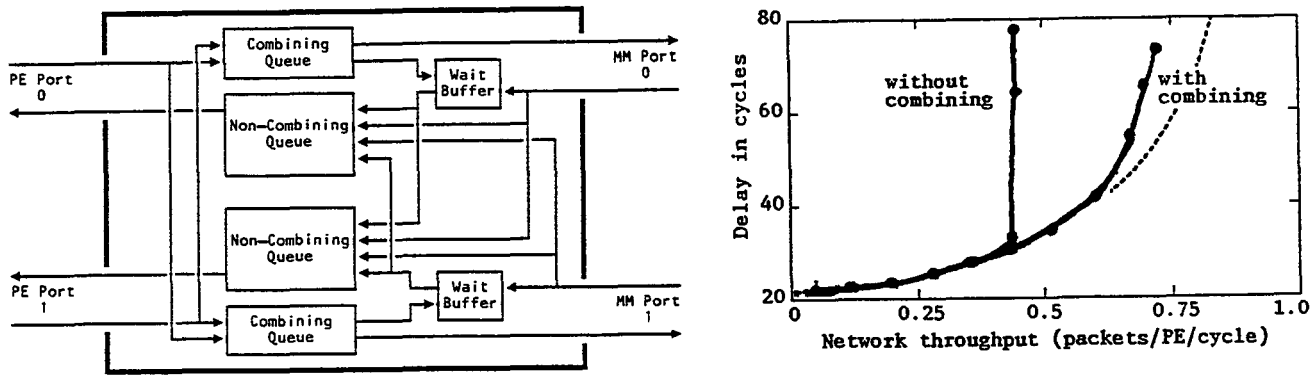


Abb. 2.1.4 Interne Struktur eines Verbindungsknotens und das Durchsatzverhalten

Parallelprogrammierung

Das Betriebssystem "Symunix" des NYU Ultracomputers ist UNIX-ähnlich und eine modifizierte Version des CMU MACH-Systems (Carnegie-Mellon University).

Der Kern des Betriebssystems des Ultracomputers ist - ebenso wie der des Hydra-Systems - sehr klein und enthält nur die notwendigsten Systemkonstrukte, um die Hardware effektiv auszunutzen. Fast alle Systemdienste sind als Benutzerprozesse gegliedert und damit auch leicht parallel auszulagern. Obwohl das Betriebssystem leicht UNIX emulieren kann, ist es völlig neu geschrieben und feinkörnig strukturiert. Wie MACH unterstützt es *messages*, *ports*, *Leichtgewichtsprozesse* und *capability-based objects*. Darüberhinaus sind folgende interessanten Erweiterungen vorgenommen worden, um das Betriebssystem parallel abarbeitbar zu machen:

- Einem Prozeß wird der Prozessor nicht zugeteilt, sondern er erledigt dies aktiv selbst mit Hilfe von Systemdiensten, die auf einer zentral gehaltenen, parallel bearbeiteten Warteschlange arbeiten. Damit ist die serielle Aktion auf kritische Abschnitte beschränkt, die durch die fetch-and-add Primitive unproblematisch werden.
- Die Speicherzuweisung (memory management) wird ebenfalls parallel vorgenommen.
- Das Ein/Ausgabesystem wurde so modifiziert, daß bei reinen Leseoperationen kein Flaschenhals auftreten kann.

Die Anwendungsprogrammierung kann auf die erwähnten Primitive zurückgreifen. Darüber hinaus wurden bei dem IBM RP3 Projekt, das in Kooperation mit der NYU sowohl die Hardwarearchitektur (allerdings mit dem IBM "RAMP" RISC Prozessor) als auch das Betriebssystem übernahmen, Präprozessoren geschaffen, die die *Parbegin* und *doall* Konstrukte (s. Abschnitt 1.3.1) als Erweiterungen der Sprachen C und FORTRAN in Makros dieser Sprache umformen. Die automatische Umsetzung ist Ziel des PTRAN-Projekts dieser Gruppe.

Fehlertoleranz

In der gesamten Hardware fehlen die systematischen Maßnahmen zur Fehlertoleranz. Da die Prozeßverteilung über eine zentrale Warteschlange abgewickelt wird, sind transiente Fehler in

diesem Bereich und in anderen zentralen Daten fatal für das System. Trotz der parallelen und unabhängigen Funktion der einzelnen Prozessoren existieren zwangsläufig Berührungspunkte wie zentrales Dispatching, I/O und dergleichen, die den Gebrauch von Konstrukten zum Gegenseitigen Ausschluß (z.B. fetch-and-add) nötig machen. Hält ein Prozessor an, der die Zugriffsrechte besitzt, so werden alle anderen Prozessoren blockiert.
Ein Ausbau der Anlage auf die geplante Größe von 4096 Prozessoren ist aber ohne Fehlertoleranzmechanismen sehr problematisch.

Als einzigen Hardware-Fehlertoleranzmechanismen ziehen Almasi und Gottlieb in [ALM] Fehlertoleranz für das Omega-Netzwerk in Betracht. Dazu kann man beispielsweise eine weitere Stufe (s. Abb. 1.2.5) in das Netzwerk einfügen, wobei alle Verbindungsknoten mit zusätzlichen, schaltbaren Überbrückungen für die Rekonfiguration ausgestattet werden. Die Fehlererkennung im Netzwerk ist allerdings dabei unberücksichtigt.

2.2 Nachrichtengekoppelte Systeme

Von den überwiegend nachrichtenorientierten Systemen werden in diesem Abschnitt zuerst sehr lose gekoppelte, LAN-basierte Systeme vorgestellt. Danach folgt als Beispiel einer festeren Kopplung nach dem "Vorzimmer"-Modell (Abschnitt 1.2) das Cm* System, und zum Abschluß die sehr stark kommunikations-orientierte, SIMD-basierte Connection machine, die auch vor dem Hintergrund des Abschnitts 4 interessant erscheint.

LAN Systeme

Auch ein lokales Netzwerk aus unterschiedlichen Rechnern läßt sich als parallel arbeitendes Rechnersystem ansehen. Hauptmerkmal eines solchen Systems ist eine Hierarchie von Nachrichtenverbindungen, die, meist räumlich verteilt, je nach Platz im Netzwerk einen unterschiedlich schnellen Zugriff auf benötigte Resourcen (Speicher, Platten, Spezialhardware) gestatten.

Hardwarearchitektur

In Abbildung 2.2.1 ist ein solches *Inhouse-LAN* abgebildet, das sich über mehrere Stockwerke erstreckt.

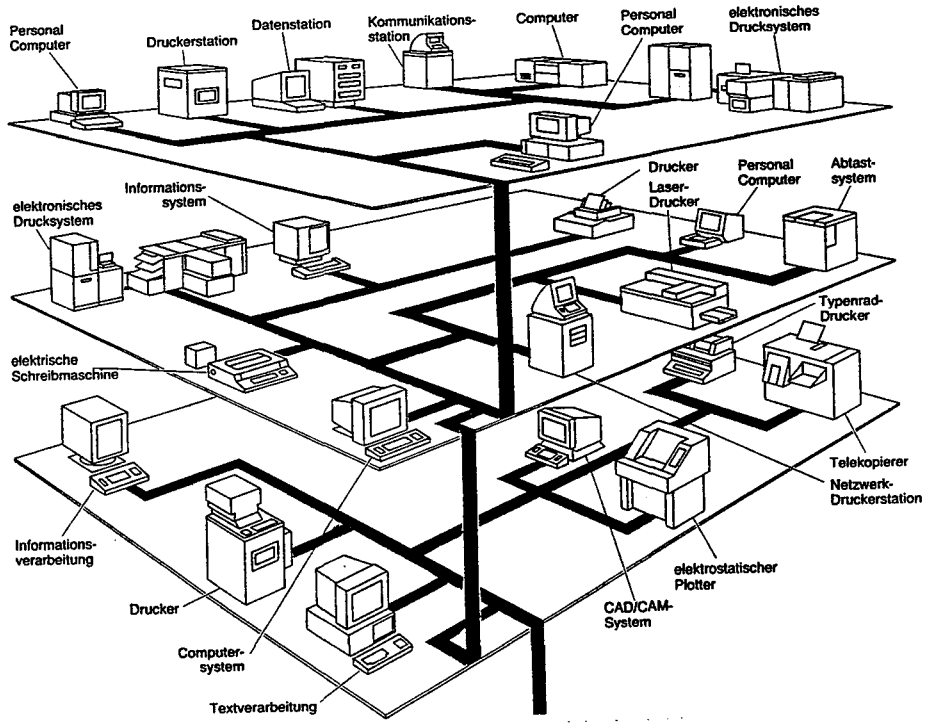


Abb.2.2.1 LAN Rechnersystem

Die Rechnerhardware in LANs ist meistens sehr inhomogen. Vorteil dieser Architektur ist es, verschiedene teure Spezialhardware wie Floating-Point Supercomputer, Datenbank-Spezialrechner, Schnelldrucker, Analog-I/O-Vorrechner und Platten- und Magnetbandstationen in ein einheitliches Zugriffsverfahren einzubinden.

Parallelprogrammierung

Obwohl die Rechner im Netzwerk parallel arbeiten können, wird diese Form der parallelen Ausführung kaum genutzt. Die Schwierigkeiten liegen darin, ein vorhandenes Programm aufzuteilen, auf den verschiedenen Maschinen zu compilieren (Betriebssystemaufrufe, Laufzeitbibliotheken!) und die Ausführung zu koordinieren. Meist ist dies nur bei Rechnern möglich, die das gleiche Betriebssystem (z.B. UNIX) und damit die gleichen Systemaufrufe verwenden. Aber auch da gibt es Schwierigkeiten, Programme auf unterschiedlichen Versionen des gleichen Betriebssystems (BSD Unix vs. System V Unix) zu verwenden.

Um die unterschiedlichen Implementierungsdetails der inhomogenen Systeme nicht berücksichtigen zu müssen, ist es deshalb sinnvoll, die Dienstleistung nur auf der Schnittstelle der Hochsprache in Anspruch zu nehmen. Dazu wird auf allen Rechnern die Möglichkeit geschaffen, eine Dienstleistung eines Rechners im Fernaufruf in Anspruch zu nehmen (*remote procedure call RPC*, s. Abschnitt 1.3.3). Dies ermöglicht dem Programmierer, mittels eines RPC in inhomogenen Systemen eine Leistung zu erbringen, die auf dem Rechner nicht möglich wäre (z.B. Datenbankabfrage) oder wesentlich länger dauern würde (z.B. "number crunching").

Erst wenn auf die Ausführung des Aufrufs nicht mehr gewartet wird, wie bei der *remote service invocation* (*RSI*), kann man von einer echten Parallelarbeit sprechen. Dies liegt beispielsweise auch dann vor, wenn ein Prozeß durch ein RPC blockiert wird und der Prozessor in der Zwischenzeit andere Prozesse bearbeiten kann.

Eine andere Form der Parallelprogrammierung stellt das Scheduling der parallelen Lastverteilung auf Jobebene in einem solchen Netzwerk dar. Im TRICE LAN-System beispielsweise existiert auf höherer Netzwerk-Managementebene MIFIA ein Koordinator, der sowohl die Filesysteme dynamisch in drei Stufen nach der Zugriffshäufigkeit ordnet, als auch auftretende RPCs nach ständig aktualisierten Tabellen über Belastung und Verfügbarkeit der Rechner im Netz umleitet und verteilt [SANT]. Damit wird nicht nur eine Lastverteilung schon beim "login" des Benutzers vorgenommen, sondern auch automatisch das Netz bei ausfallenden Rechnern transparent für den normalen Benutzer rekonfiguriert.

Fehlertoleranz

Die nachrichtengekoppelten Systeme besitzen meist ein hohes Maß an Rekonfigurierbarkeit, sobald ein Fehler entdeckt und lokalisiert worden ist: Die Protokollschichten der Kommunikation (s. Abschnitt 1.4.2) erlauben vielfach ein für den aufrufenden Anwenderprozeß transparentes Erkennen, Korrigieren und Rekonfigurieren bei Ausfall der Kommunikationswege. Begrenzt wird diese Form der Fehlertoleranz nur durch die relativ geringe Zahl der Kommunikationswege (z.B. multiple Busse) und die meist nur geringe Redundanz (begrenzte Anzahl) der Spezialhardware (Ausfall des Superrechners, Ausfall des I/O-Rechners).
Die Verantwortung für die korrekte Funktion der einzelnen Rechnerkomponente im Netz liegt allerdings bei dem Rechner selbst, so daß für eine effektive Fehlertoleranz hier mindestens Fail-Stop-Prozessoren (s. Abschnitt 1.4.1) vorliegen sollten.

Cm*

Mit dem Cm* (*C*omputer *m*odule, "sehr oft vorhanden") Projekt wurde versucht, als Nachfolgesystem des C.mmp (s. voriger Abschnitt) ein Computersystem mit Hilfe von vielen Grundmodulen nach der "Vorzimmer"-Architektur zu realisieren [SWAN] und die sich daraus ergebenden Charakteristiken für parallele Algorithmen zu erkennen [JON].

Hardwarearchitektur

Die Hardware besteht im Wesentlichen aus der Verbindung von Mikrocomputern (DEC PDP11/03 mit max. 250kB Speicher), die zu je einem Cluster von 10 Modulen mittels eines *map bus* gekoppelt wurden. Insgesamt 5 dieser Cluster wurden über einen Controller (*K.map*) und einen Inter-Cluster-Bus zu dem Gesamtsystem zusammengefaßt. Der map bus controller (BusControl, *S.local*) in jedem Mikrocomputer entscheidet, ob eine Adressreferenz vom Prozessor lokal zum PDP-11-Q-Bus oder global über den map bus zum K.map-Controller geleitet wird. Der K.map-Controller und der S.local Schalter formen zusammen also eine Memory-Management-Unit. In Abbildung 2.2.2 ist die Gesamtarchitektur gezeigt.

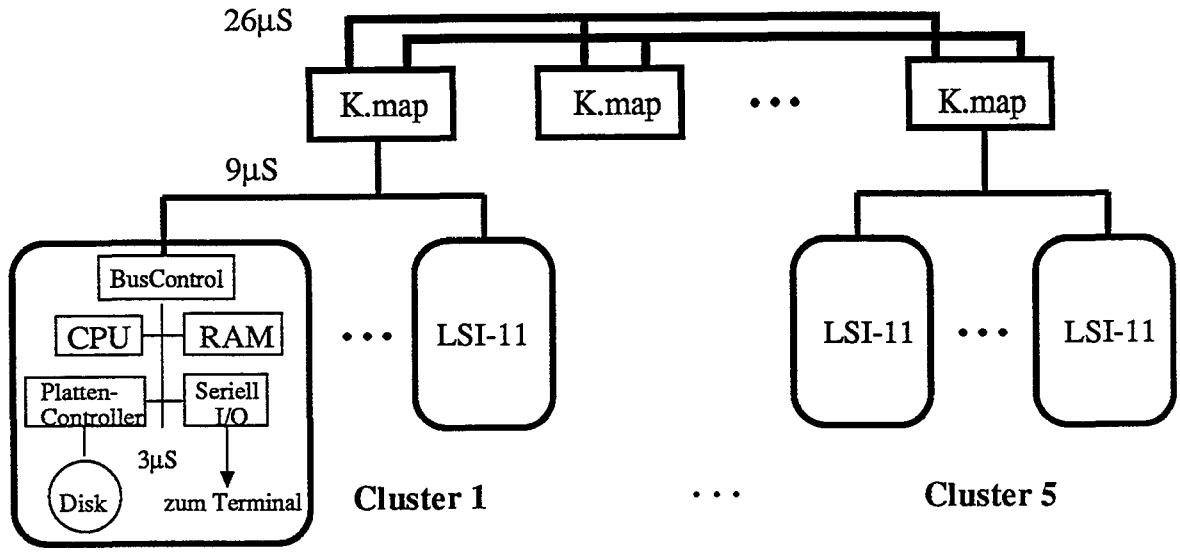


Abb. 2.2.2 Das Cm* Netzwerk

Obwohl der lokale Speicher für jeden Prozessor direkt zugreifbar ist und die Architektur von Cm* mit einem globalen, mit 28 Bit byteadressierbaren Adressraum deshalb vielfach als *shared memory* Design angesehen wird, existiert doch eine Hierarchie in den Zugriffszeiten (1:3:9, s. Abb.), die eine Partition der Software nach nachrichtenorientierten Gesichtspunkten (s.u.) nahelegt. Damit wurde das System als "nachrichtenorientiert" klassifiziert, obwohl es durch die Hardwaremöglichkeiten zwischen den fest und lose gekoppelten Systemen steht.

Die globale Adressierung wird durch die K.map-Einheit unterstützt, die die zu Datenpaketen zusammengefaßten Speicherzugriffe puffern kann und den gegenseitigen Ausschluß beim Zugriff auf Speicherstellen sicherstellt.

Parallelprogrammierung

Das Betriebssystem "StarOS" wurde als nachrichten- und objektorientiertes Hilfsmittel dem Hardwaresystem "maßgeschneidert" [JON]. Ebenso wie Hydra sind die meisten Funktionen (Speicherallozierung, I/O, etc) als durch Nachrichten kommunizierende Benutzerprozesse ausgeführt. Den Kern des Betriebssystems (ca. 6kB und 2500 Mikrocode-Befehle) bilden die Adressierungsoperationen, Kommunikationsprimitive, Synchronisierung sowie Stack-und Warteschlangenoperationen.

Ein ungelöstetes Problem bildet das Benutzer-Programmiermodell des Computers (*virtueller Computer*). Macht man es zu fein, so ist der Benutzer unnötigerweise mit vielen Details konfrontiert, die zu Fehlern führen können. Wird dagegen zuviel der Computerhardware "transparent" für den Benutzer, so können schlechte Ausführungszeiten resultieren, deren Ursachen dem Benutzer unklar bleiben.

Schon bald zeigte sich beim Studium von parallelen Algorithmen, daß es nicht zu vernachlässigen ist, auf welchen Prozessoren das Programm und in welchem Speicher sich die nötigen Daten befinden. Als beste Strategie zeigte sich diejenige, die den Stack sowie alle lokalen Daten dem lokalen Speicher zuordnete und alle globalen Variablen im betreffenden Cluster hielt. In Abbildung 2.2.3 sind die Ergebnisse von verschiedenen Konfigurationen für das Lösen eines Systems von Differentialgleichungen aufgetragen.

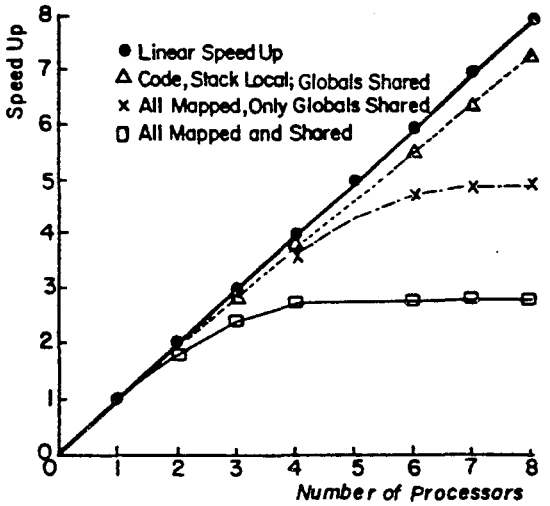


Abb. 2.2.3 Speicherkonfigurationen und Ausführungsbeschleunigung von partiellen Differentialgleichungen

Aus diesem Grunde unterstützte das StarOS sowohl Lowlevel- Operationen (Zuweisen von Daten an einen bestimmten Speicher) als auch höhere Konzepte, die algorithmische Datenabhängigkeiten auf den Zugriffs"abstand" der Architektur abbildete. Dazu gehörte die Unterstützung von ununterbrechbaren Leichtgewichtsprozessen (*chores*), von Gruppen von chores in Modulen, die mittels Nachrichten Funktionen ausführen (*Objektkonzept*), von Prozessen und ihrer Kommunikation und von Gruppen von Prozessen (*task forces*), die einem Job zugeordnet (*group scheduling*) sind. Um dies effektiv programmieren zu können, wurde eine besondere Scheduling Sprache TASK und ein Compiler geschrieben, der eine optimale Abbildung der "Zugriffsnähe" der Objekte auf die Zugriffshierarchie der Hardware leisten und die Datenplazierung automatisieren soll.

Fehlertoleranz

Die Fehlerlokalisierung im Cm* System hatte einige Schwierigkeiten, die durch den verwendeten Prozessortyp bedingt waren. Brach beispielsweise ein Prozessor den Betrieb mit einer *power fail trap* ab, so konnte trotz stabilen Speichers danach nicht weitergerechnet werden, da der stack pointer mit dem Programmzähler beim "Hochfahren" hardwaremäßig überschrieben wurde.
Die Rekonfiguration des Systems war nicht automatisiert und konnte nur per Hand vorgenommen werden.
Außer den Standardmöglichkeiten, die nicht über die Konzepte von C.mmp hinausgingen, gab es keine speziellen Fehlertoleranzmöglichkeiten auf dem Cm* System.

Die Connection Machine

Als ein interessantes Beispiel einer Rechnerarchitektur, die sehr viele Prozessoren in einem nachrichtenorientierten Verbindungsnetzwerks eines Hypercubes (s. Abschnitt 1.2) einsetzt, zeigt sich die Connection Machine CM-1 [HILL]. Sie wurde ursprünglich dazu entwickelt, konnektionistische, symbolorientierte Modelle und Programme (s. Abschnitt 4.3.1) effizient auszuführen. Bald jedoch zeigten sich als Hauptanwendungen der Einsatz in physikalischen Problemen (Finite Elemente, ray tracing etc, s. Abschnitt 4.4.1), was zu einem Redesign der Maschine führte. Bei dem Nachfolgemodell CM-2 wurde nicht nur das I/O-System verbessert, sondern auch Fließkomma-Coprozessoren eingebaut, was zu einer theoretischen Höchstleistung von 2,5 GFLOPS führte.

Die Hardwarearchitektur

Im Unterschied zu den vorher vorgestellten Systemen enthält die CM sehr viele, sehr einfache Prozessoren - im Vollausbau 65.536 Stück - und eine hohe Vernetzung, wie es für konnektionistische Systeme typisch ist. Die 1 Bit-Prozessoren werden im SIMD-Modus mit *Nanoinstruktionen* geladen, die von einem Host-kontrollierten Mikrocontroller aus den Maschineninstruktionen (*Macroinstruktionen*) der Maschine erstellt werden.

Die Maschine besteht aus 4096 an den Microcontroller direkt angeschlossenen Einheiten (Knoten), die direkt mit 12 Nachbareinheiten zu einem Hyperwürfel verbunden sind. Jede dieser durch eine 12Bit-Adresse festgelegten Knoten besteht aus einem Chip, auf dem 16 Prozessoren, ein Instruktionsdekoder und ein Router integriert sind, sowie 16 × 4kBit externer statischer Speicher.

Eine zweite, schnellere Kommunikationsmöglichkeit besteht durch eine 1-Bit Verbindung zu je einem Nachbarprozessor in Nord-, Ost-, Süd- und Westrichtung, die nicht nur auf dem Chip besteht, sondern auch zu den vier Nachbarchips. Dadurch sind alle Prozessoren in einem gitterartigen Kommunikationsnetz eingebunden.
In der Abbildung 2.2.4 ist das Schema eines Knotens verdeutlicht.

Jeder der 1-Bit Prozessoren besteht wiederum nur aus einem 16Bit-Statuswort und einer ALU, die zusammen mit dem externen Speicher ausschließlich Instruktionen der Form

$$OP\ (A,B,S_1) \rightarrow A,\ S_2$$

verarbeiten kann. Dabei werden bei den Operationen (z.B. ADD und AND) ein Bit der als Register A und B fungierenden, externen Speicherzellen mit einem Statusbit verknüpft und sowohl das Register A in einem Nachbarn als auch ein eigenes Statusbit neu gesetzt. Jede Nanoinstruktion vom Macrocontroller enthält neben der 16 Bit Wahrheitstabelle, die die ALU-Funktion (und damit die

Prozessorinstruktionen) definiert, die beiden 12-Bit Adressen für A und B, die beiden 4-Bit Adressen für die Statusbits sowie 2 Bits für die Bestimmung des Nachbarn und 4 Bit für ein Bedingungsbit des Statusregisters, das die individuelle Ausführung der Instruktion veranlasst oder untersagt. Damit ist es möglich, nicht jede Instruktion auf allen Prozessoren ausführen zu lassen.

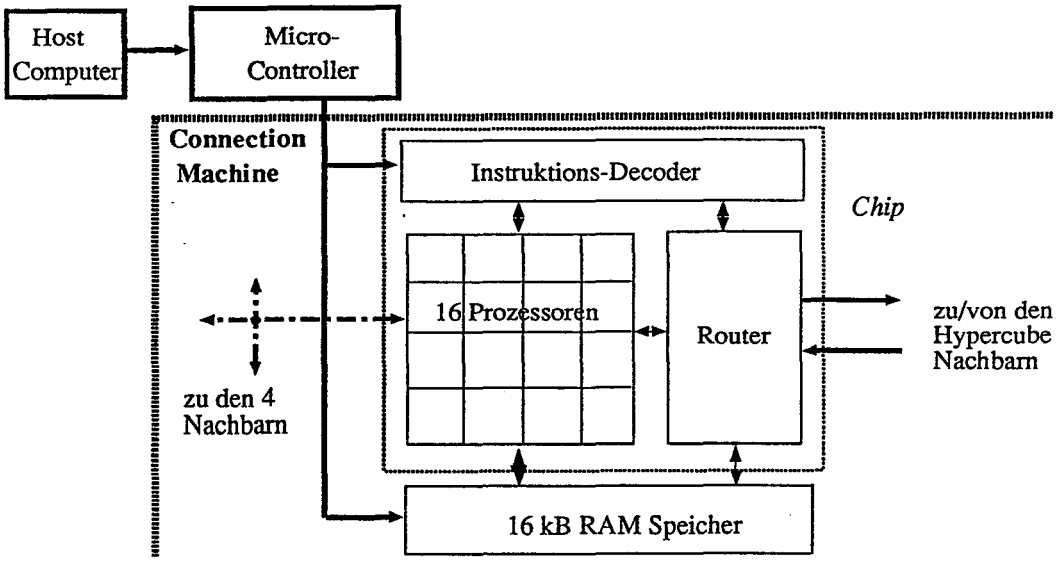


Abb. 2.2.4 Ein Knoten der Connection Machine

Insgesamt sind 32 der 64-Pin Prozessor-Chips (also 512 Prozessoren) mit ihren Speicherchips auf einer Steckkarte montiert, von denen 16 Stück in einen 0,75m breiten Metallgitterkasten eingesteckt sind. Der gesamte Computer besteht aus 8 dieser Kästen, so daß alle 65.536 Prozessoren in einem Würfel von nur 1,5m Kantenlänge Platz finden.

Parallelprogrammierung

Die Programmierung der parallelen 1-Bit-Operationen wird durch die Existenz von Lisp, C und FORTRAN Compilern erleichtert. Die Menge der 1-Bit-Daten wird zu Vektoren ("xector") zusammengefaßt und elementare Datenoperationen darauf definiert, die durch die Compiler unterstützt werden.

Beispielsweise wird ein α-Operator ("α-notation") definiert, der die Addition zweier Vektoren, beispielsweise der Datentupel {1, 2} und {3,4}, erlaubt. In CmLisp hat dies die Form

$$(\alpha + \text{'}\{a\rightarrow 1\ \ b\rightarrow 2\}\text{'}\ \{a\rightarrow 3\ b\rightarrow 4\})$$

wobei die Notation $\rightarrow$ eine Initialisierung der xector-Komponente ausdrückt.

Die Summe (Betragsquadrat) eines Vectors wird durch eine β-Operation zur Verfügung gestellt, die in CmLisp für den Vektor {3,4} lautet

$$(\beta + \text{'}\{a\rightarrow 3\ \ b\rightarrow 4\})$$

Die Addition läßt sich leicht komponentenweise lokal durch die Prozessoren durchführen; die Summation durch Verschieben (routing) der Daten zwischen den Prozessoren. Die zur Durchführung der Summation nötige Kommunikation zwischen den Daten wird damit auf die Kommunikation zwischen den Bits der Daten, also auf die Kommunikation zwischen den Prozessoren abgebildet.

Dabei spielt es aber durchaus eine Rolle, ob die Kommunikation nur zwischen den Prozessoren eines Chips oder mit erheblich höherem Aufwand zwischen zwei Prozessoren verschiedener Chips stattfindet. Deshalb machen die meisten schnellen Algorithmen keinen Gebrauch von dem Hypercube-Netzwerk.

2.3 Das ATTEMPTO System

Das ATTEMPTO System ist ein Multi-Mikroprozessor System, bei dem versucht wurde, sowohl den Belangen der parallelen als auch der fehlertoleranten Ausführung von Programmen Rechnung zu tragen. Da der Autor an der Konzipierung und Implementation dieses Systems über längere Zeit mitgewirkt hat, soll dieses System im Folgenden als konkretes Beispiel dienen, um die prinzipiellen Probleme der Parallelausführung und Fehlertoleranz in Multi-Mikroprozessorsystemen im Detail zu diskutieren.

Der Aufbau von ATTEMPTO sollte dabei in erster Linie die Tragfähigkeit der von uns entwickelten Konzepte zur Fehlertoleranz und Parallelität validieren sowie als Testbett die Erprobung anderer Parallelitäts- und Fehlertoleranzkonzepte ermöglichen und weniger einen marktreifen Prototypen eines Produktionssystems darstellen. Trotzdem entwickelten sich die Vorstellellungen über ein wünschenswertes, fehlertolerantes Multiprozessorsystem an den industriellen Gegebenheiten: Obwohl die Notwendigkeit fehlertoleranter Rechner heute unumstritten ist, weisen aber die wenigen, bis jetzt vorhandenen, industriellen Produkte aus unserer Sicht noch immer erhebliche Mängel auf:

- Die Fehlertoleranzmechanismen müssen vom Anwender in seinen Programmen berücksichtigt werden (kostspielige Programmänderungen!); Standardsoftware von dritter Seite ist deshalb kaum vorhanden
- Die erarbeitete Lösung ist meist sehr speziell, nicht modular und damit auch nicht portabel
- Die benutzte Hard- und Software entspricht keinem industriellen Standard
- Das System ist (relativ zur nicht-fehlertoleranten Version) einfach zu teuer

2.3.1 Die Anforderungen an das System

Im Gegensatz zu den Industrielabors war unsere Arbeitsgruppe in der Situation der Anwender. Als wir begannen, zur Erprobung unserer Fehlertoleranz-Konzepte den fehlertoleranten Rechner ATTEMPTO zu entwickeln, entschieden wir uns für folgende allgemeine Systemvorgaben [BR6]:

1) Der Anwender soll das System als ein "normales", multi-tasking Monoprozessor System sehen. Dies bedeutet insbesondere, daß die Fehlertoleranzmechanismen für den Anwender transparent sein sollen, d.h. alle, auch absolut gebundene Programme sollten ohne Änderung fehlertolerant ablauffähig sein.

2) Der Anwender kann entscheiden, wie er die Rechenleistung zwischen Fehlertoleranz und Durchsatz aufteilt.

3) Die Hardware-Komponenten müssen Standardteile, keine Spezialanfertigungen sein.

4) Die Software (Betriebssystem, Utilities) muß Standard sein.

5) Die nötige Zusatzsoftware (z.B. für die Parallelarbeit und Fehlertoleranzmaßnahmen) soll portabel sein, d.h. die Zusatzsoftware ist modular und strukturiert in einer höheren Programmiersprache zu schreiben.

2.3.2 Hardwaredesign

Um die Auswirkungen von Defekten möglichst gering zu halten, entschlossen wir uns zu einem modularen Hardwaredesign, das nicht nur eine gute Rekonfigurierbarkeit erlaubt, sondern auch eine einfachere hardwaremäßige Lokalisierung und Isolierung der Defekte. Damit kamen nur nachrichtenorientierte Systeme in Betracht, die aus Effizienzgründen möglichst die gesamte benötigte Software (Programm und Betriebssystem) im lokalen Speicher der Prozessoren enthalten sollten (vgl. Erfahrungen bei Cm* im vorigen Abschnitt).

Für die Hardware-Module entschieden wir uns nach der oben erwähnten dritten Vorgabe, Single-Board-Computer (SBC) an einem Multi-Master Bus (s. Abschnitt 2.3.7) einzusetzen. Zwar wäre aus Effizienz- und Fehlertoleranzgründen ein multiples Bussystem besser geeignet (s. Abschnitt 1.2), aber zum Zeitpunkt der Hardwarebeschaffung war ein solches nicht verfügbar. Allerdings sahen wir beim Softwaredesign bereits die Möglichkeit vor, das Kommunikationssystem dafür zu erweitern.

Da das Betriebssystem ebenfalls Standard sein sollte (Vorgabe 4), wurde UNIX dafür gewählt. Sämtliche zusätzlich nötige Software ist, soweit unbedingt für den UNIX-Kern nötig, in 'C' geschrieben; der Hauptteil allerdings in der höheren, fehlervermeidenden Programmiersprache (Vorgabe 5) Modula-2 [WIR].
Eine Übersicht über die gewählte Hardware-Konfiguration ist in Abbildung 2.3.1 gezeigt.

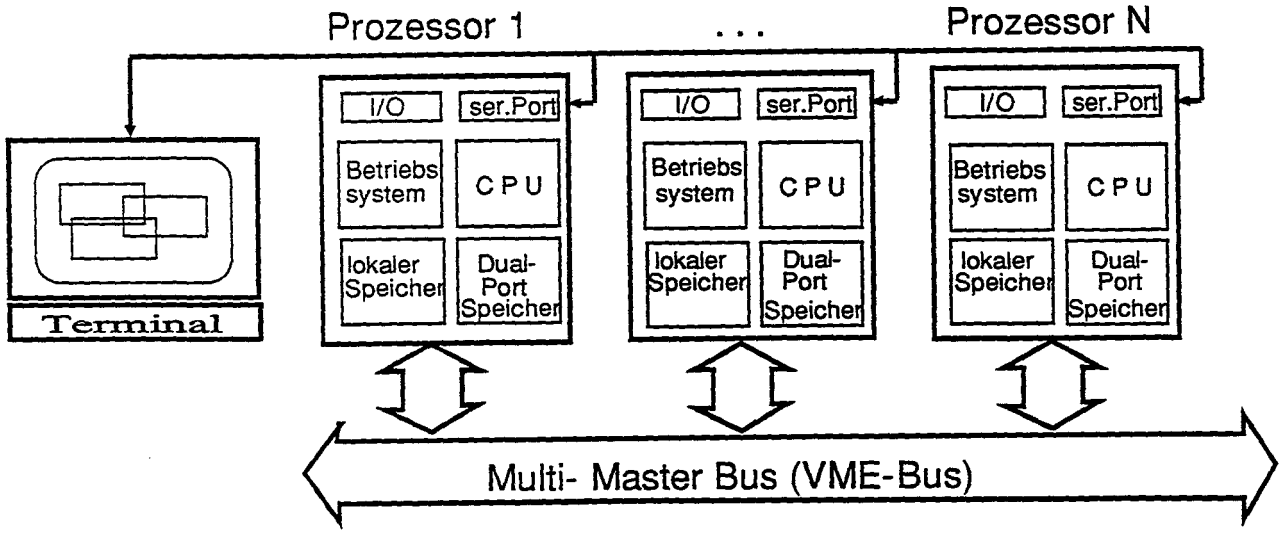


Abb. 2.3.1 Die Hardwarekonfiguration des ATTEMPTO 1 Systems

Obwohl der lokale Speicher der SBCs auch über globale Adressierung über den Bus zu erreichen ist, wird der Speicher nicht als globaler Speicher genutzt. Der Verbindungsaufbau zwischen den beteiligten Prozessoren und den erreichbaren Speichern ist durch die Existenz des Dual-Port Speichers eine Mischung zwischen der "Tanzsaal"- und der "Vorzimmer"-Systematik aus Abschnitt

1.2.: Über den Multi-Master Bus ist sowohl eine Interprozessor-Kommunikation als auch eine Prozessor-Speicher-Verbindung möglich.
Nähere Einzelheiten zur Hardwareimplementierung sind im Abschnitt 2.3.7 zu finden.

2.3.3 Softwaredesign

Im folgenden Abschnitt wird - ausgehend von den Systemanforderungen in Abschnitt 2.3.1 - das Design und die Struktur der Software beschrieben, die aus diesen Anforderungen folgen. Betrachten wir zunächst dazu das System aus der Sicht des Benutzers.

Die Benutzersicht von ATTEMPTO

Der Benutzer sieht den Arbeitsplatzrechner gemäß Anforderung 1 als Single-User, Multi-Tasking-System; die Realisierung als Multi-Prozessor-System ist ihm verborgen. Das System bietet dennoch die Möglichkeit, die Fehlertoleranzeigenschaften im Gegensatz zu [WEN2] individuell für jede Anwendung zu wählen. Beispielsweise spezifiziert ein Eintippen von "MEINPROGRAMM #2#", daß "MEINPROGRAMM" mit dem Fehlertoleranzgrad t=2 ausgeführt werden soll. Dies bedeutet, daß bei der Ausführung des Programms maximal t Defekte (und damit transiente oder permanente Fehler auf maximal t SBC) in dem Sinne toleriert werden, daß keine fehlerhaften Ergebnisse ausgegeben werden.

Dabei kann der Benutzer das System so viele Programme gleichzeitig ausführen lassen, wie es die Systemkapazität erlaubt: mehrere Programme mit geringem oder wenige mit hohem Fehlertoleranzgrad.

Systemübersicht

Das Betriebssystem von ATTEMPTO besteht aus identischen, autonomen, lokalen Betriebssystemen *ATOS* (*ATTEMPTO's local Operating System*) und gliedert sich in zwei funktionale Schichten. Die untere Schicht wird von einem UNIX-ähnlichen Einprozessor-Mehrprozeß-Betriebssystem gebildet. Darauf baut eine Zwischenschicht FTL (*Fault Tolerance Layer*) auf, welche die parallele Ausführung der Benutzerprogramme (*UserJob*) und die Fehlertoleranzeigenschaften (Vergleich der Ergebnisse der UserJobs im Fehlertoleranzbetrieb) bereitstellt und für das Benutzerprogramm transparent ist (s. Abbildung 2.3.2). Vom Betriebssystemkern wird sie als normaler, hoch-prioritärer Prozeß behandelt [BR4].

Mit der Einführung der Zwischenschicht ist es nun möglich, auch der schwierigen Vorgabe 1) aus Abschnitt 2.3.1 zu genügen. Um die Ein- und Ausgabe des Benutzerjobs zu Fehlertoleranz-zwecken kontrollieren zu können, werden deshalb alle UNIX-Systemaufrufe (*SysCalls*) des Benutzerjobs nicht direkt ausgeführt, sondern vorher von Teilen der Zwischenschicht überprüft.

Umleitung der System-Calls

Bei einem Betriebssystem-Aufruf (*SysCall* bzw.*trap*), angefordert von einem UNIX-Prozeß bzw. einem Benutzerjob, wird statt der vorgesehenen Behandlungsroutine im Betriebssystem-Kern eine Prozedur *CIkernel* im Kommunikationsteil CI aufgerufen. Diese entscheidet, ob zur weiteren Behandlung des Aufrufes eine Instanz der FTL aktiv werden muß oder ob der SysCall direkt an das Betriebssystem übergeben werden kann. Insbesondere sind dies folgende SysCalls unter UNIX:

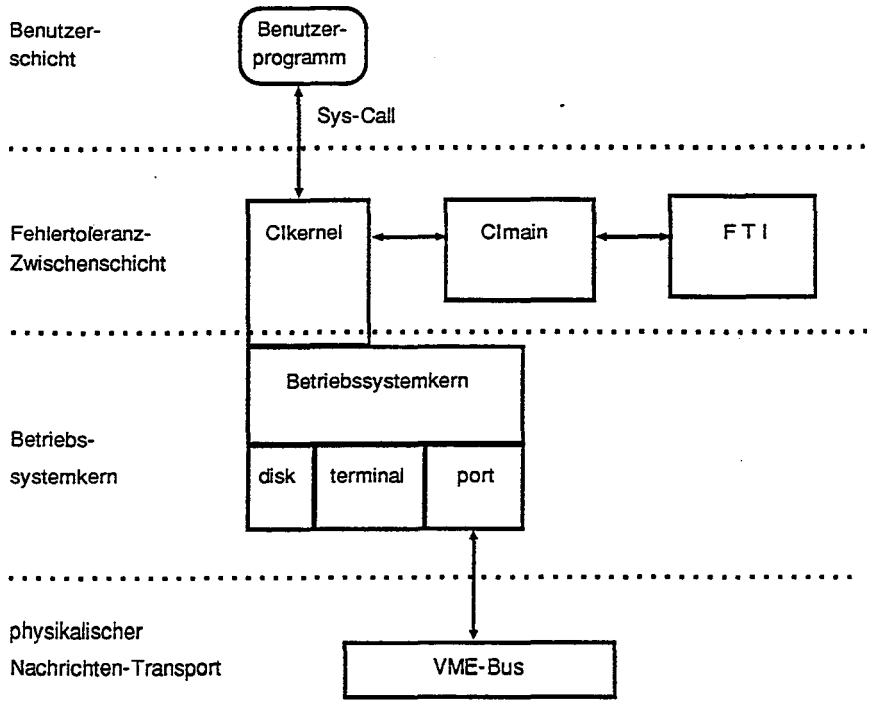


Abb. 2.3.2 Die Einbettung der Zwischenschicht in UNIX

- *open()* und *close()* einer globalen Resource (z.B. Terminal)
- *read()* und *write()* für globale Resourcen
- *fork()* um die Prozeßnummer der Kinder für read/write zu gewinnen
- *exit()* um Kenntnis von der Terminierung des Benutzerjobs und seiner Kinder zu erlangen

Falls Dienstleistungen der FTL für die Behandlung des SysCall nötig sind, wird dies in Form einer Nachricht über ein Modul *PostOffice* an entsprechende Teile der FTL angestoßen. Nach der Bearbeitung innerhalb der FTL geht dann die Kontrolle an die Routine CIkernel und damit an den aufrufenden UNIX-Prozeß zurück.
Dieses Umleitungsverfahren bietet verschiedene Vorteile:

- jedes ablauffähige Programm kann nach Maßgabe des Benutzers auch fehlertolerant abgearbeitet werden.
- Betriebssystem-Kerne sind in der Regel vor unzulässigen (beabsichtigt oder nicht) Eingriffen der Benutzer geschützt. Dieser Schutz erstreckt sich damit auch auf den Zugang zur FTL.
- da nur an eine einzige Stelle im Kern eingesprungen wird, bleibt die Änderung lokal und kontrollierbar.
- auch ohne die Kenntnis der Quellen des Betriebssystem-Kernes ist es möglich, eine Zwischenschicht einzufügen. (So können z.B. die Parameter der System-Calls anhand des jeweils zugehörigen, rückassemblierten Codes verifiziert werden.)

ATOS ist bis auf die Routine *CIkernel* maschinenunabhängig. Da jedoch trap-Mechanismen auf weitgehend allen Mikroprozessoren vorhanden sind, läßt sich das Umleitungskonzept leicht auf andere Prozessor-Typen übertragen. Die entsprechende Routine CIkernel ist an jeden

Betriebssystem-Kern (z.B. als pseudo-device) anzulagern.

Für die Kommunikation zwischen den SBC-Prozessoren existiert ein Nachrichtenaustausch-mechanismus, der über sog. Ports (s. Abschnitt 2.3.4) realisiert ist und in UNIX als *special file* eingebunden ist. An das Betriebssystem wird für die Inter-Prozessor-Koordination nur die Anforderung gestellt, daß Nachrichten in der gleichen zeitlichen Reihenfolge an die Fehlertoleranzschicht weitergegeben werden, in der sie an das Betriebssystem von den Device-Handlern übergeben werden (s.Abschnitt 2.3.4).

Die Fehlertoleranzschicht FTL

Zusätzlich zu dem für die Fehlertoleranz zuständigen, als Fehlertoleranzinstanz FTI (*Fault-Tolerance Instance*) bezeichneten Teil der FTL gibt es noch die Kommunikationsinstanz CI (*Communication Instance*), die für die Umleitung der SystemCalls und Transferieren ihrer Parameterdaten zuständig ist. Um die FTL-Module möglichst uniform, einfach und testbar zu halten, sind die Ein- und Ausgabeinformationen in einem einheitlichen Nachrichtenformat strukturiert. Für die Eingabe der Daten von den externen Datenquellen wie

- Terminal (Benutzer-Eingabe)
- Devices (Magnetplatte etc.)
- Daten des Benutzerjobs über die CI, ausgelöst durch einen System Call
- Nachrichten aus dem Kommunikationsport von anderen FTL anderer SBC

gibt es allerdings Schwierigkeiten. Will man zur Verarbeitung der Ausgabedaten des UserJob die Zwischenschicht als UNIX-Prozeß danach aktivieren, so muß zur Vermeidung von *Deadlocks* bzw. *Pufferüberlauf* die FTL höhere Priorität beim UNIX-Dispatcher haben als jeder andere UserJob. Alle kausal vom UserJob angestoßenen Aktionen müssen als Nachrichten bei der FTL verarbeitet worden sein, bevor die nächste Aktion des UserJob bearbeitet werden kann. Da die Daten der externen Geräte asynchron eintreffen, kann sich die FTL nicht auf eine bestimmte Dateneingabe (Datenkanal) zum Warten festlegen; eine wartende (blockierende) Leseoperation auf einem Kanal würde andere, wichtige Daten auf anderen Kanälen blockieren. Benutzt man aber eine nicht-blockierende, repetative Abfrage aller möglichen Datenkanäle (Polling), so kommt es durch die hohe Priorität der FTL zu einem *Life-Lock*: durch die Abfrageaktivität der FTL können keine neuen Daten vom UserJob erzeugt werden.
Eine Lösung für dieses Problem stellt die Übertragung des Wartens der FTL auf dedizierte Stellvertreterprozesse dar. Jedem Eingabekanal wird ein eigener UNIX-Prozeß zugeordnet, der seperat von der FTL blockierend auf Daten seines Kanals wartet. Treffen neue Daten ein, so liest dieser Prozeß die Daten, formatiert sie in das einheitliche Nachrichtenformat und leitet sie durch eine *pipe* an die FTL weiter. Durch die Verwendung des UNIX *pipe*-Konstrukts [BEL] werden die Nachrichten sowohl gepuffert als auch ihre zeitliche Reihenfolge erhalten. Das Schreiben in eine pipe ist atomar, d.h. es kann nicht von einem Prozeß-Wechsel unterbrochen werden. Dadurch sind die Nachrichten immer zusammenhängend und die FTL kann die Daten der verschiedenen Nachrichtenquellen eindeutig separieren und verarbeiten.

Obwohl CIkernel als Teil des Betriebssystemkerns beim Durchlaufen des SystemCall zum UserJob gerechnet werden muß, ist es sinnvoll, für die Nachrichten an die FTI einen extra

UNIX-Prozeß CIK vorzusehen. Dies ist durch folgendes Problem bedingt:

> Das Schreiben in eine pipe geschieht in UNIX ausschließlich vom Programmspeicher des UserJob (*user space*) aus. Da vor den eigentlichen Daten aber noch der Nachrichtenkopf geschrieben werden muß, müßte für das Absenden einer Nachricht in einem Schreibvorgang (*atomic action*) extra Speicherplatz im Benutzerprogramm reserviert und die Nachricht dort zusammenkopiert werden. Dies bedeutet aber einen Eingriff in das Benutzerprogramm, der sich mit der Forderung nach Transparenz der Fehlertoleranzmechanismen (Kapitel 2.3.1) nicht verträgt.
>
> Stattdessen wurde die Lösung gewählt, den Filedeskriptor des UserJob so zu ändern, daß er dem Filedeskriptor einer pipe zum CIK Prozeß entspricht. Diese pipe wird bei der Initialisierung des Prozeßsystems eingerichtet und ihr Filedeskriptor im erweiterten UserRecord des UNIX Prozesses transparent für den UserJob bei jedem "fork()" mit weitergegeben. Bei einem "write()" wird der Header direkt und die Daten durch die CIK-pipe zum CIK Prozeß übertragen. Der CIK setzt dann beide zusammen und überträgt sie als integrale Nachricht atomar über die pipe zur FTI.

Diese Designentscheidung hat auch noch den Zusatzvorteil, daß die Entscheidungsinstanz über die Umleitung der SysCalls modular auf CIkernel bzw. CIK beschränkt bleibt und damit schnelle Reaktionen ermöglicht. Außerdem bietet sich damit eine Möglichkeit an, die asynchron durch den UserJob erzeugten Daten einer Flußkontrolle zu unterwerfen, um ein Datenüberlauf bei der Pufferung in der FTI zu verhindern.

In Abbildung 2.3.3 ist die UNIX-Prozeßstruktur gezeigt. Die UNIX-Prozesse und die pipes zur Kommunikation werden bis auf die shell SH bzw. den Benutzerjob UJ alle beim Starten des Systems (*booten*) erzeugt. Die shell bzw. der Benutzerjob werden immer dann von der FTL erzeugt, wenn der Benutzer einen neuen Auftrag gegeben hat und durch Kommunikation mit den anderen SBC abgestimmt wurde, daß gerade dieser SBC den Job übernimmt (Prinzip der Auftragsanziehung, siehe Abschnitt 2.3.5).

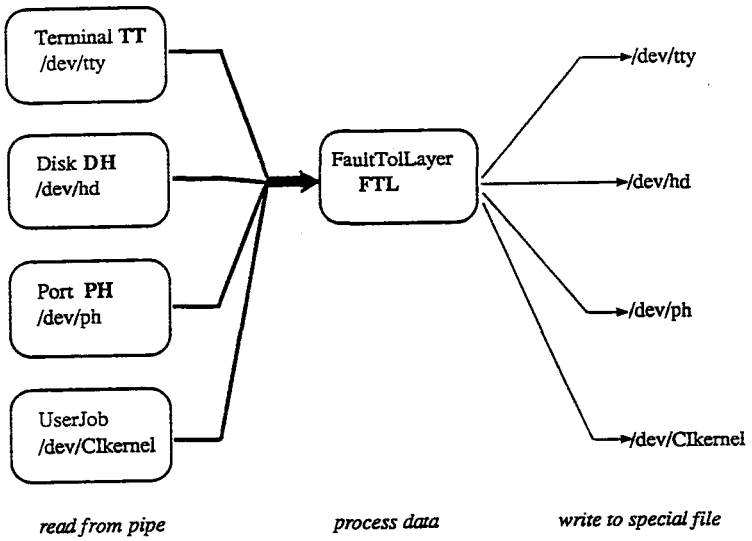


Abb. 2.3.3 Das UNIX-Prozeßsystem der Zwischenschicht FTL

Die verschiedenen Funktionen der in Modula-2 programmierten Zwischenschicht FTL wie Job-dispatching (FTD), Pufferverwaltung (DIB,DOB), Fehlertoleranzaufgaben (SAB) und Verwaltung der globalen Resourcen (RM) werden als eine Anzahl von kooperierenden Leichtgewichtsprozessen (s. Abschnitt 1.3.2), basierend auf dem Modula-2 Coroutinenkonzept, realisiert.
In Abbildung 2.3.4 ist ein Überblick über die Gesamtstruktur der FTL zu sehen.
Betrachten wir nun im Folgenden die Funktionen der FTL etwas näher.

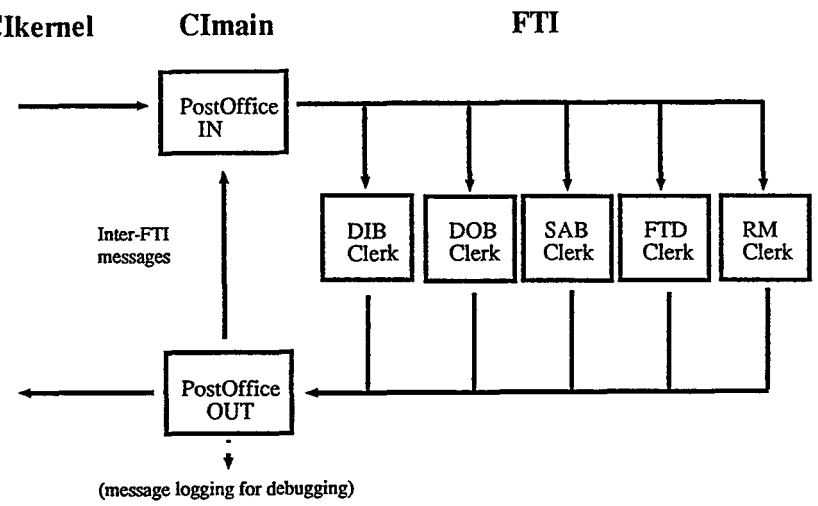


Abb. 2.3.4 Die Modula-2 Leichtgewichtsprozesse der FTL und ihr Datenfluß

Die Kommunikationsinstanz CI

Wenn ein Benutzerprogramm (*UserJob*) eine Dienstleistung des System anfordert (z.B. Eingabe/ Ausgabe), so geschieht dies durch einen SysCall an das zugrunde liegende Betriebssystem. Die vor den SysCall handler (trap-handler) geschaltete Kommunikationsinstanz bewirkt eine Umleitung der System-Calls zu der eingeschobenen Zwischenschicht.

Die Kommunikation zwischen UserJob und FTL kann nicht über gemeinsame Speicherbereiche erfolgen, da inaktive UNIX-Prozesse auf Massenspeicher ausgelagert werden können. Sie wird stattdessen über sogenannte pipes abgewickelt, d.h. über gemeinsame Prozeß-unabhängige Speicherbereiche, die dem direkten Zugriff entzogen sind und nur über spezielle System-Calls angesprochen werden können. Dazu teilt sich die CI auf in einen dem Betriebssystemkern angehefteten Teil *CIkernel*, in dem der Unix-Prozeß 'UserJob' die System-Calls via CIK in kurze Nachrichten eines festen Formats umformt und die Daten (Parameter des SystemCalls) in eine pipe zum Unix-Prozeß 'FTL' schreibt, und dem entsprechenden Teil *CImain* der FTL, der diese Nachrichten liest, die Daten aus der pipe entnimmt und beides dem *PostOffice* zur fehlertoleranten Ausführung der Aktionen des Benutzerprogramms (lesen, schreiben, etc) übergibt.

Neben der Prozedur CImain gehört auch das für die FTI zentrale Kommunikationsmodul PostOffice zur CI, das die zwei eigenen Clerks (Modula-2 Leichtgewichtsprozesse) *POin* und *POout* enthält. Der POin-Clerk wirkt als Briefverteiler: er wird durch Nachrichten in seinem Briefkasten aktiviert, liest den Empfänger und legt die Nachricht in dessen Briefkasten ab. Der POout-Clerk ergänzt die Nachricht eines FTI-Clerks um die für die Inter-Prozessor-Kommunikation nötige Information und reicht die Botschaft in Form eines direkten System-Aufrufes (CI-Call) über den Betriebssystem-Kern an den entsprechenden Porthandler weiter (s.Abb.2.3.4).

Das Modul PostOffice bildet also die Schnittstelle zwischen FTL und dem Betriebs-system-Kern und stellt die Weiterleitung von Information zwischen FTL und Benutzerjob, Port- bzw. Terminalhandler und dem File-System sicher. Das PostOffice wird genau dann aktiv, wenn es von einem Clerk der FTI durch deren Botschaft beauftragt wurde oder wenn es eine Nachricht vom Porthandler-Prozeß, vom Terminalhandler-Prozeß oder von CIkernel erhält. Das PostOffice verwendet dazu verschiedene Prozeduren, die in ihrer betriebssytem-spezifischen Implementierung im Modul 'CImain' zusammengefaßt sind.

Die Umleitung von System-Calls kann damit wie in Abbildung 2.3.5 skizziert werden.

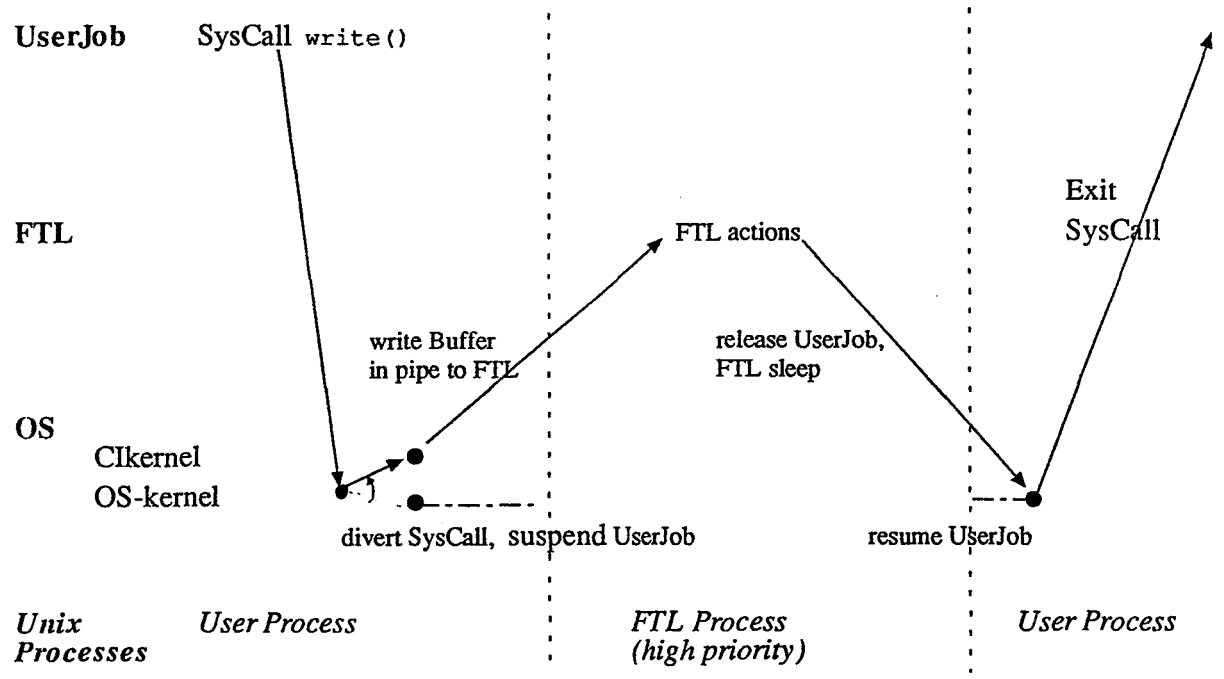


Abb. 2.3.5 Umleitung von SysCalls über PostOffice zur Bearbeitung in der FTL

Die Fehlertoleranzinstanz (FTI)

Die lokale FTI ist verantwortlich für folgende Betriebssystemaufgaben:

- lokale Interpretation der Benutzerkommandos an ATTEMPTO
- Management der systemweiten Resourcen (z.B.Terminal)
- Kontrolle der Daten von und zu den Ein- und Ausgabegeräten
- Systemweite Konsistenz der Systemtafeln

Außerdem werden von der FTI folgende Funktionen für Fehlertoleranzzwecke bereitgestellt (s. Abschnitt 3.4.2) :

- dezentrales Dispatching der Benutzerprogramme
- Vergleich der Ausgabedaten der lokalen Kopien eines Benutzerprogramms und anschließende Diagnose bei Nichtübereinstimmung
- Überwachung bei der Ausgabe der Daten
- Fehlerinterpretation und -behandlung

Aus Gründen der Fehlertoleranz besitzt jede FTI dazu ihre eigene Systemtafel, die alle aktuellen Angaben über Kollegen zur Programmausführung, anstehende Benutzerprogramme, Zustand der globalen Resourcen sowie aller SBC im System enthält.

Die Software-Architektur der Fehlertoleranz-Schicht FTL basiert also auf einer vierstufigen Hierarchie (s. Abb.2.3.6), deren oberste Schicht 4 (FTI) die eigentlichen Fehlertoleranz-Mechanismen enthält. Schicht 3 (CI) ist im wesentlichen für die korrekte Kommunikation zwischen Benutzerjobs und FTI sowie -auf logischer Ebene - zwischen dem Host-SBC und den restlichen

SBCs des Systems zuständig. Die beiden unteren Schichten stellen Dienstleistungen zur Verfügung, die für die Verwaltung der Modula-2-Prozesse (Coroutinen), der Datenstrukturen und des Botschaftenaustausches von den darüberliegenden Schichten benötigt werden.

			Schicht	
prozeßorientiert	*gekapselte Datenobjekte*	DIB, DOB, FTD, SAB, RM	4	FTI
	Kommunikation	TOutClock, PostOffice, CImain	3	CI
prozedurorientiert	*Datentypen und -prozeduren*	iProcesses, Messages, Resources, Strings, Lists	2	
	Systemschicht	SYSTEM, SysCalls, AStorage	1	

Abb. 2.3.6 Hierarchie der Fehlertoleranz-Schicht in ATOS

Dementsprechend gibt es zwei verschiedene Arten von grundlegenden Software-Bausteine in der FTL :

- Module mit einer strikt prozeduralen Schnittstelle, die meist abstrakte Datentypen sowie die darauf definierten Prozeduren implementieren (Schicht 2) und
- Module, die Datenstrukturen zusammen mit einer aktiven Einheit, dem sogenannten Clerk (einem Leichtgewichtsprozeß), enthalten (Schicht 4).

Die Datenstrukturen der letztgenannten Module sind von außerhalb der Module nicht zugänglich (*objektorientierte Datenkapselung*). Sie werden vielmehr allein von den entsprechenden Clerks verwaltet. Die Kommunikation zwischen den Clerks verschiedener Module geschieht durch Botschaftenaustausch (s.Abb.2.3.4). Dieses Prinzip erleichtert vor allem die Rekonfigurierbarkeit der für die Fehlertoleranz des Systems zuständigen obersten Schicht (vgl. dazu auch [LIS]).
Eine Zwischenstellung nehmen die Module der Schicht 3 ein. Auch hier wurde nach Möglichkeit das Prinzip der Datenkapselung verwirklicht.
Das Modul "Messages" der Schicht 2 stellt den Clerks Prozeduren für den Botschaftenaustausch bereit, die auf "Signalen" des Modules "iProcesses" beruhen. Daneben werden Dienstleistungen, z.B. die Erzeugung von Briefkästen erbracht.
In ATOS sind drei Ebenen der Kommunikation zu unterscheiden:

- Kommunikation zwischen Clerks (Modula-2 Prozesse)
- Kommunikation zwischen UNIX-Prozessen
- Kommunikation zwischen SBC-Prozessoren

Für den Botschaftenaustausch zwischen Clerks ist jedem Clerk ein eigener Briefkasten zugeordnet. Die FTL-lokale Übergabe einer Botschaft erfolgt per Referenz, um unnötiges Kopieren zu ersparen. Die Inter-Prozeß-Kommunikation in UNIX folgt den UNIX-Konventionen. Sie kann über pipes mittels eines *memory device* als pipe device abgewickelt werden.

Dagegen erfolgt der Nachrichtenaustausch zwischen SBC's über sogenannte dedizierte Ports auf der Grundlage eines speziellen Protokolls, das in Abschnitt 2.3.4 näher beschrieben wird. Zur Verwaltung dieser Ports wurde auf jedem SBC ein sogenannter *Porthandler PH* eingerichtet.
In Abbildung 2.3.7 ist ein Überblick über den Nachrichtenverkehr zwischen den Clerks auf einem SBC und systemweit zwischen den SBCs zu sehen.

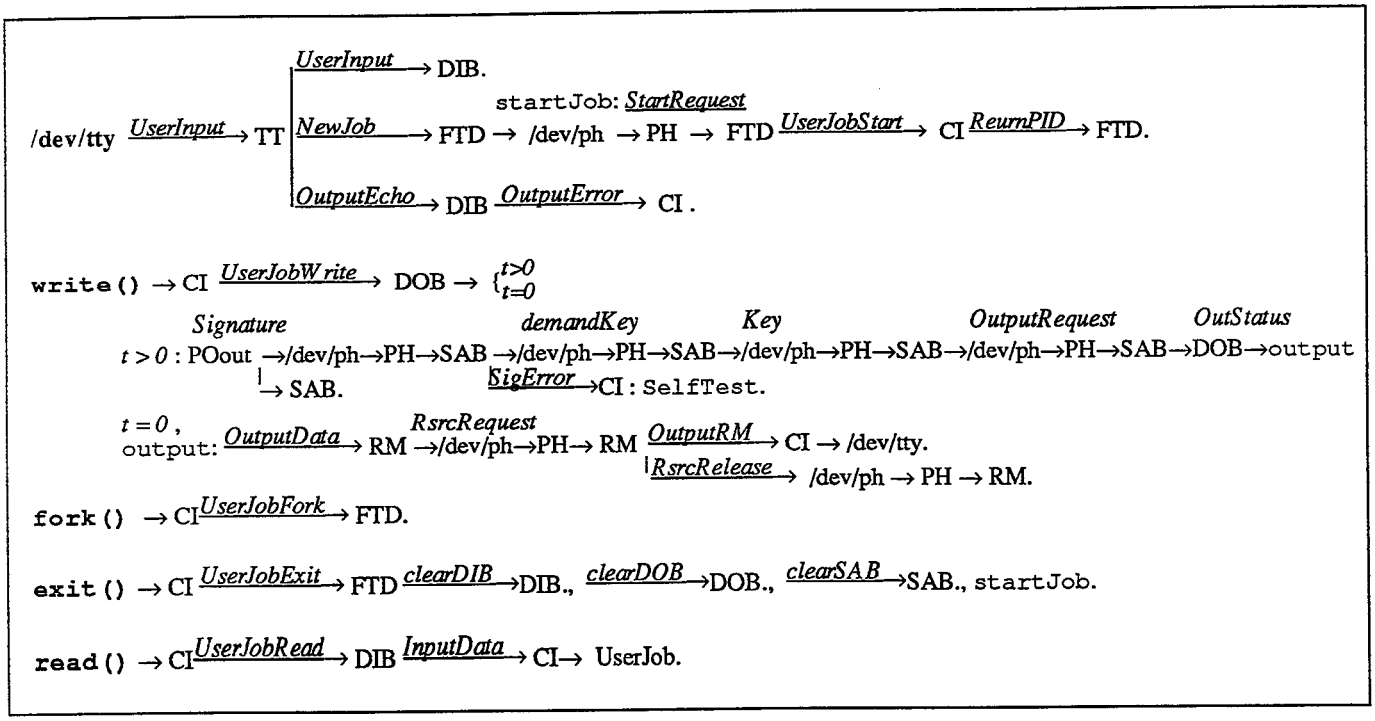


Abb. 2.3.7 Die Nachrichtenfolgen in ATTEMPTO

Alle Briefkästen, Daten-Puffer, Warteschlangen usw. sind als Ausprägungen eines einzigen abstrakten Datentyps Lists implementiert. Um diese zentralen Datenstrukturen unempfindlich gegen Speicherfehler zu machen, wurde das Modul "Lists" konzipiert, das den abstrakten Datentyp List zur Verfügung stellt. Inhärenter Bestandteil der Listenstruktur ist die Möglichkeit, eine beschränkte Anzahl von Fehlern in der Zeigerstruktur zu erkennen und zu korrigieren [RIS3].

Im Folgenden wollen wir die verschiedenen aktiven Bestandteile der FTI genauer betrachten, sofern sie für die Organisation des Multiprozessorsystems nötig sind. Die speziellen Funktionen für die Fehlertoleranz sind dagegen erst in Abschnitt 3.4.2 näher ausgeführt.

Die Datenpuffer

Die Benutzereingaben werden vor der FTI im Terminalprozeß TT (s. Abb. 2.3.3) in Kommandos, Eingaben für den aktuellen UserJob sowie in Ausgabenechos unterteilt. Die Kommandos werden dem Fehlertoleranz-Dispatcher FTD (s.u.), die Eingaben dem DIBClerk (s.u.) und die Ausgabenechos zur Prüfung dem DOBClerk (s.u.) zugeleitet.

Obwohl die Eingabe direkt an alle SBC herangeführt ist (s. Abb.2.3.1) und immer nur ein Job von einem SBC bearbeitet wird (s. Abschnitt 2.3.5) kann sich eine zeitliche Verzögerung zwischen der Eingabe des Benutzers auf einem Terminal und der Eingabe-Aufforderung des UserJob ergeben. Um dem Benutzer eine erneute Eingabe der selben Daten zu ersparen,werden alle Inputdaten in einem Data Input Buffer (DIB) von dem DIBClerk verwaltet.

Da zwischen dem Erzeugen der Ausgabedaten und deren tatsächlicher Ausgabe bei fehlertolerantem Betrieb eine Phase des Vergleichs und der Abstimmung mit den anderen SBC-Kollegen liegt, muß die Ausgabe zwischengepuffert werden. Für die Verwaltung des Puffers gibt es einen Leichtgewichtsprozeß, den Data Output Buffer (DOB) Clerk. In Abbildung 2.3.8 sind

seine Reaktionen sowie die der anderen Clerks auf verschiedene Nachrichten verdeutlicht.

MODULE DIB	
Datenstruktur:	Data Input Buffer DIB, Puffer für vom Benutzer eingegebene Daten (für READ des Benutzerjobs)
aktive Einheit:	DIBclerk, verwaltet den Eingabe-Puffer
Botschaften:	
UserInput:	Eintragen der Input-Daten in den Puffer.
UserJobRead:	Untersuchung, ob angeforderte Daten im Puffer vorhanden sind. Falls vorhanden, Versenden einer Botschaft mit Kennung 'InputData' über PostOffice an Benutzerjob, sonst Versenden einer Botschaft mit Kennung InputRequest' an DOBclerk.
clearDIB:	Aushängen aller zum angegebenen Job gehörenden Input-Daten.
initDIB:	Initialisierung des DIBuffer
MODULE DOB	
Datenstruktur:	Data Output Buffer DIB, Puffer für die auszugebenden Daten vom UserJob
aktive Einheit:	DOBclerk, verwaltet den Ausgabe-Puffer
Botschaften:	
UserJobWrite:	Falls Anzahl der Kollegen = 1, Ausgabe des entsprechenden Pufferinhalts; andernfalls Übermittelung des entsprechenden Pufferinhaltes an SABclerk in Botschaft mit Kennung 'OutputData'
OutStatus:	Wenn IoMaster, dann Beauftragen des Resource-Managers mit Ausgabe des entsprechenden Puffer-Eintrags, sonst Überwachen der Ausgabe des jeweiligen IO-Masters
InputRequest:	Ausgabe der Aufforderung 'Input für JobID' an den Benutzer wie oben beschrieben
clear DOB:	Aushängen aller zum angegebenen Job gehörenden Output-Daten
init DOB:	Initialisierung des DOBuffers

Abb. 2.3.8 Die Datenpuffer- Module und ihre Funktionen

Die Zeitschranken

Wichtige Nachrichtenfolgen werden über Zeitschranken überwacht. In gewissen Fällen wird dann die entsprechende SBC-Einheit als defekt angesehen. Die Problematik von Termin-Überschreitungen im Zusammenhang mit Fehlertoleranz wird schon in [WEN] aufgezeigt (vgl. auch [LAM]).
Auf der FTI-Ebene werden im Wesentlichen die Zeit zwischen Auftragsvergabe (*NewJob*) und Auftragsannahme (*StartRequest*), die Zeit zwischen Erzeugen von Ausgabedaten (*OutputData*) und Empfang der letzten Signatur, die Zeit zwischen dem Verlangen des Ausgabeschlüssels (*DemandKey*) und dem Eintreffen desselben und schließlich die Zeit zwischen dem Sperren einer Resource (*ResRequest*) und dem Freigeben (*ResRelease*) über Time-Out abgesichert. In der folgenden Abbildung wird das Modul kurz charakterisiert.

MODULE TimeOut	
Datenstruktur:	ClockList : Liste von Verbunden mit Timeout-Zeitpunkt und Zeiger auf solche Botschaften, die mit Zusatzinformation 'TimeOut' über Postoffice wieder an den Sender zurückgegeben werden, sobald das Timeout- Intervall abgelaufen ist (vermittels der Routinen setTimeOut und clearTimeOut)
aktive Einheit:	ClockClerk, überprüft regelmäßig ClockList nach Botschaften, deren TimeOut-Intervall abgelaufen ist,und gibt diese zurück
Botschaften:	keine

Abb. 2.3.9 Das TimeOut- Modul

Es gibt aber auch Zeitbedingungen der im OSI Modell (s. Abschnitt 1.4.2) unteren Kommunikationsschichten, die im Port-Handler (s. Abschnitt 2.3.4) beachtet werden müssen.

ATOS enthält außer den hier beschriebenen Modulen das Modul "FTLinit" für die Initialisierung der FTL sowie einige Datei-Module, in denen allgemein verwendete Konstanten und Typen definiert sind. Dadurch wird vermieden, daß diese Objekte aus Modulen der oberen Schichten importiert werden müssen. Dies soll die Portierung von ATOS erleichtern.

Resourcenverwaltung RM

Das Modul *Resources* dient der Verwaltung der Betriebsmittel, die systemweit von allen Prozessoren benutzt werden können und deren Benutzung deshalb in Multiprozessorsystemen global geregelt werden muß. Dazu besitzt der Resource-Manager eine Liste aller globalen Resourcen sowie eine Zustandstafel, in der eingetragen ist, ob die Resource gesperrt ist oder nicht. Globale Resourcen sind beispielsweise der Terminal-Bus, der Drucker, etc. Die lokalen Massenspeicher (Winchester-Festplatten, s. Abschnitt 2.3.7) der einzelnen SBC werden zunächst einmal als Erweiterung des lokalen RAM betrachtet und unterliegen somit weder der Fehlertoleranzverwaltung noch der globalen Resourcenverwaltung. Mit diesem Ansatz ist die Benutzung des lokalen Filesystems, beispielsweise der Zugriff auf Zwischenfiles eines Compilers, ohne Performance-Verluste möglich.

Zweifelsohne wird aber auch ein Filesystem benötigt, das fehlertolerant beschrieben wird, beispielsweise beim Umlenken der Terminalausgabe eines fehlertoleranten Jobs auf einen File. Dies wird durch die Existenz einer Partition jeder Winchester bewerkstelligt, die als *global* gekennzeichnet und beim ResourceManager eingetragen ist. Jeder Zugriff auf diese Partition unterliegt somit der Zugriffskontrolle, und im Fehlertoleranzbetrieb der üblichen Datenüberprüfungsprozedur (s. Abschnitt 3.4.3). Diese Mechanismen reichen allerdings nicht aus, um das Filesystem als *fehlertolerant* zu kennzeichnen. Ein fehlertolerantes Filesystem, das beispielsweise auch den Ausfall und die Rekonfiguration von Massenspeichern toleriert (vgl. [BORG]) , benötigt noch weitere Eigenschaften und ist Gegenstand heutiger Forschungen. Unter dem Benutzungsmodell aus Abschnitt 3.4.1 soll es deshalb im Folgenden nicht weiter betrachtet werden.

Im Unterschied zu einem Monitor (s. Abschnitt 1.3.2) existiert bei einem Nachrichten-orientierten System wie ATTEMPTO 1 aus Fehlertoleranzgründen für die Systemtafel kein von allen Prozessoren benutzter, gemeinsamer Datenbereich; jeder SBC und damit jede Fehlertoleranz-Zwischenschicht besitzt ihre eigene Systemtafel globaler Resourcen. Damit ist es nötig, diese Daten global konsistent zu halten. Im nachfolgenden Abschnitt 2.3.4 wird ein Konzept eines 'atomaren Broadcast' vorgestellt, mit dem dieser Forderung Rechnung getragen werden kann.
Dabei kann man von einem Protokoll ausgehen, das erst dann wieder eine neue Resource-Anforderung von einem RM vorsieht, wenn die Resource vom benutzenden RM zurückgegeben wurde oder der parallel aufgesetzte Time-Out ausläuft. Dieses Protokoll führt aber bei stark interaktiven, nicht-fehlertoleranten Programmen (beispielsweise einem Editor) zu unverhältnismäßig großem Verwaltungsaufwand und Nachrichtenverkehr (Overhead) bei jedem Zeichen, das ausgegeben werden soll. Dies läßt sich vermeiden, indem man das Protokoll folgendermaßen abändert: Eine Resourcenbelegung wird erst dann zeitüberwacht, wenn eine andere Resourcenanforderung vorliegt. Dies gestattet einem interaktiven Programm beim Fehlen von Ausgabewünschen anderer Programme, das Terminal einmal zu belegen und dann für die Dauer der Sitzung zu benutzen, ohne auf eine Zeitüberwachung Rücksicht nehmen zu müssen.

2.3.4 Die Inter-Prozessor Koordination und Synchronisation

Um einen Systemzusammenbruch bei Defekt des Systembusses oder des gemeinsamen Speichers zu verhindern, hat in ATTEMPTO jeder SBC seine eigene Systemtafel, die er durch Nachrichtenaustausch mit denen der anderen SBCs konsistent hält. Dabei tritt aber das Problem auf, daß die Verarbeitung einer Nachricht selbst wieder vom Zustand der Systemtafel abhängt; so kann beispielsweise ein Prozessor nur dann für einen Job in die Systemtafel eingetragen werden, wenn die Kollegenliste noch nicht vollständig ist. Eine Rückfrage führt in diesem Fall zu erhöhter Kommunikation und belastet das System zusätzlich.

Konsistenz der Systemtafeln

Wie kann man die Systemtafeln trotzdem auf allen SBC konsistent halten ?

Anstatt nur eine, zentrale, fehleranfällige Systemtafel zu schaffen, konzipierten wir eine andere Lösung: *Die zeitliche Reihenfolge der Nachrichten zur Veränderung der Systemtafeln ist bei allen SBCs gleich zu halten.* Damit sind bei gleichem initialem Ausgangszustand die Systemtafeln auch ohne Rückantwort jederzeit konsistent.

Diese Idee ist sehr ähnlich der unabhängig davon an anderer Stelle im gleichen Zeitraum entwickelten Konzeption für die Kontrolle von verteilten Datenbanken und wird dort als *Serialisierung von Transaktionen auf verteilten Daten* [BERN] bezeichnet. Sie setzte sich als ein Grundprinzip konsistenter Datenhaltung in verteilten Systemen durch und ist eine der Voraussetzungen, um *atomic broadcast* und damit *atomare Transaktionen* (s. Kapitel 1.3.3) auf den globalen Daten durchzuführen.

Wie läßt sich dieses Konzept der Nachrichtenreihenfolge, verbunden mit den anderen Eigenschaften (endliche Zeit und Atomizität der Übermittlung) von *atomic broadcast*, möglichst effektiv mit der vorhandenen Kommunikationshardware umsetzen, ohne auf den Zeitstempel eines zentralen Transaction-Managers oder den zentralen seriellen Transaktionsgraphen wie in Datenbanken vertrauen zu müssen?

Nachrichten-Austausch

Eine Möglichkeit, die zeitliche Reihenfolge der Nachrichten auf allen SBCs identisch zu halten, bietet ein Broadcast-Bus. Bei dieser Lösung wird jede Nachricht von allen SBCs im System empfangen. Da dieser Bus von verschiedenen Prozessoren nur nacheinander benutzt werden kann, ist damit eine eindeutige Reihenfolge der Nachrichten gegeben. Das Senden einer Nachricht entspricht dabei einer atomaren, ununterbrechbaren Aktion auf dem Broadcast-Bus.

Vom Standpunkt der Fehlertoleranz ist ein solcher Broadcast-Bus aber nicht unbedenklich. Es ist dabei nicht möglich, zwischen zwei Prozessoren Nachrichten auszutauschen, ohne daß ein im System befindlicher, defekter Prozessor diese lesen oder verfälschen kann. Es wäre deshalb besser, ein Kommunikationssystem zu verwenden, bei dem der Sender nur dem eine Nachricht übermittelt, der sie auch erhalten soll.

Da außerdem in den meisten handelsüblichen Bussystemen (z.B. in dem VME-Bus unserer Implementation, s. Abschnitt 2.3.7) für eine Hardware-Broadcast-Eigenschaft eine zusätzliche, nicht-triviale Hardware-Änderung auf jeder SBC-Platine nötig wäre, entschieden wir uns im Einklang mit der Vorgabe, nur Standard-Hardware zu verwenden, die für eine Serialisierung der

globalen Transaktionen (s.[BERN]) nötige Broadcast-Eigenschaft mit den vorhandenen, allgemeinen Hardware-Elementen zu realisieren.
Dazu benutzten wir die Eindeutigkeit der Adresskennung auf den Bussystemen (s. Abb. 2.3.10).

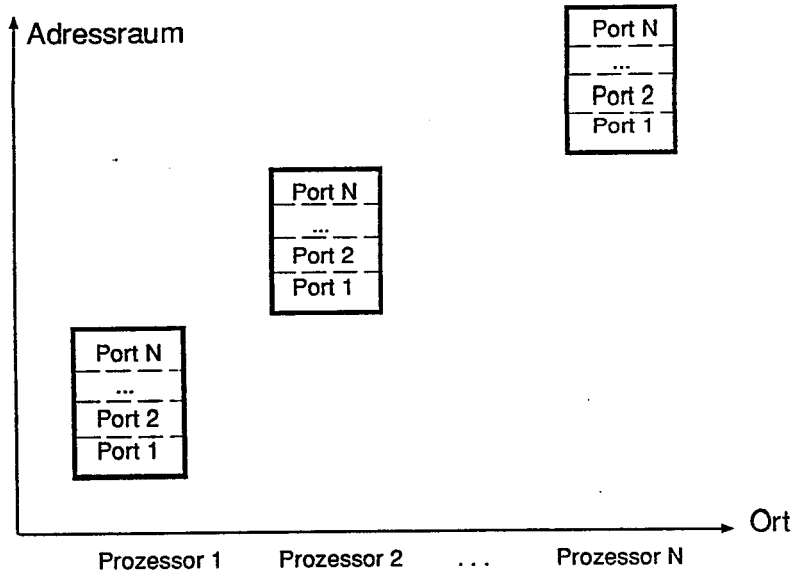


Abb. 2.3.10 Aufteilung des globalen Adressbereichs

Der Nachrichtenaustausch der Interprozessor-Synchronisation und Kommunikation wird in ATTEMPTO über *ports*, spezielle Speicherbereiche des Dual-Port-RAM auf den SBCs, abgewickelt. Auf jedem SBC existiert ein Port für den Empfang von Nachrichten von jedem SBC des Systems. Damit wird zwischen jeweils zwei Prozessoren eine dedizierte, logische Verbindung eingerichtet; die Prozessoren sind also logisch voll vernetzt [BR9].

Eine Besonderheit ist die Verbindung zu sich selbst, mit der auf elegante Weise die Konsistenz der Systemtafeln eingerichtet wird: Jede Nachricht, die globale Daten bezüglich einer Gruppe von Prozessoren enthält, muß als "Broadcast-Nachricht" an alle Prozessoren dieser Gruppe verschickt werden. In diesem Fall wird die notwendige gleiche Reihenfolge aller Nachrichten auf allen Boards durch die Maßnahme erreicht, *allen* Mitgliedern der Gruppe *gleichzeitig* die Nachricht zu senden - auch sich selbst.

Dieses Koordinationsprinzip läßt sich zusätzlich auch auf die Time-Out-Ereignisse anwenden, die in der Fehlertoleranz-Zwischenschicht auf einem SBC initiiert werden (s. Kapitel 2.3.3). Alle time-out Ereignisse werden vom TimeOutClerk als Nachrichten über den port an sich selbst geschickt. Damit ergeben sich gleich zwei Vorteile: über den Sendemechanismus der unteren Protokollschichten wird nicht nur ausgeschlossen, daß die Ursache des time-out beim eigenen SBC liegt, Nachrichten nicht richtig empfangen zu können (Test der Kommunikation), sondern auch, daß die erwartete Nachricht noch ungelesen in der Empfangs-Warteschlange liegt, obwohl sie rechtzeitig angekommen ist. Die Bemessung der time-out Zeiten wird damit unabhängig von der Auslastung des Empfangs-SBC.

Der sichere Austausch von Daten wird durch eine höhere Busbelastung erkauft - nämlich dann, wenn eine Nachricht an mehrere zu versenden ist. Da jeder Prozessor sofort feststellen kann, ob er eine neue Nachricht erhalten hat, entfällt dafür im Unterschied zu einem Broadcast-Bus die Notwendigkeit, jede einzelne Nachricht zu lesen, um den Empfänger festzustellen.

Koordination des Nachrichtenaustauschs

Allerdings stellt sich ein weiteres Problem: wenn eine Nachricht von einem Prozessor an alle anderen geht, darf das Versenden einer Nachricht an mehrere Empfänger nicht durch das Senden eines anderen Prozessors unterbrochen werden, da sonst die Empfangsreihenfolge der beiden Nachrichten auf allen SBCs nicht mehr identisch ist.

Eine übliche, aber nicht fehlertolerante Möglichkeit, dieses Problem zu lösen, ist die Sperrung des Busses während der gesamten Sendedauer einer Nachricht an alle Empfänger. Selbst die Absicherung einer solchen Bussperrung durch einen Watch-dog timer, wie dies auf vielen Boards üblich ist, kann aber nicht die dauerhafte Blockade des Busses durch einen defekten Prozessor verhindern.

Um die Fehlerquelle eines blockierten Kommunikationsbusses zu vermeiden, entschieden wir uns für folgendes Konzept:

- Jedem Prozessor am Kommunikationsbus ist eine Interrupt-Signalleitung zugeordnet, die er aktiviert, nachdem er seine Nachricht allen Empfängern gesendet hat.

Die Reihenfolge, in der die Nachrichten vom lokalen ATOS registriert werden, ist somit nicht vom Eintreffen der Nachrichten auf den verschiedenen SBCs bestimmt, sondern von der Reihenfolge der dazugehörigen Interrupts.

Das gleichzeitige Senden der Nachricht an alle Prozessoren beim Broadcast wird also hier durch ein überall gleichzeitig anliegendes Interruptsignal ersetzt. Im Vergleich zu den üblichen Broadcast--Systemen ist für den Nachrichtenaustausch bei diesem Konzept eine Interruptleitung für jeden SBC nötig.

Die Forderung, auf allen SBCs die gleiche zeitliche Reihenfolge der Nachrichten zu garantieren, wird damit zu der Forderung, auf allen SBCs die Bearbeitung der Interrupts in der gleichen Reihenfolge sicherzustellen. Im Folgenden werden vier Probleme geschildert, die bei der Realisierung dieser Forderung auftreten, und Lösungsmöglichkeiten angegeben.

a) Fehlende Interruptsequenz - Hardware

Jede Interrupt-Service-Routine (ISR) benötigt eine gewisse Mindestdauer, z.B. die Zeit T_{SYNC} zum Umladen der Register und Initialisieren der Routine. Erfolgen innerhalb dieser Zeitspanne erneut Interrupts, so können diese nicht sofort bearbeitet werden, sondern werden in einem Interruptregister als Ereignis gespeichert. Diese Register erlauben aber keine Aussage mehr über das zeitliche Eintreffen der Ereignisse.

Würde man statt der konventionellen nun spezielle Hardware entwickeln, die auch die zeitliche Reihenfolge der Interrupt-Ereignisse beachtet (Aufbau einer FIFO), so wäre es trotzdem nicht möglich, die Unterschiede der fertigungsmäßig und thermisch bedingten Offset-Spannungen der Interrupteingänge sowie die Unterschiede in den Signallaufzeiten zwischen den verschieden weit entfernten SBC zu beseitigen. Dies bedeutet für zwei gleichzeitig, aber an verschiedenen Orten generierte Interrupts, daß sie je nach Offset und Signallaufzeiten früher oder später registriert und damit unterschiedlich eingeordnet werden.

Das Problem der identischen zeitlichen Reihenfolge läßt sich mit Hilfe der Standard-Hardware befriedigend lösen, indem man den Interruptleitungen der Kommunikation Prioritäten zuordnet. Dies impliziert eine feste, andere Ordnung der Abarbeitung der Interrupts als die der

zeitlichen Reihenfolge. Da diese Ordnung aber auf allen SBCs identisch ist, ist die Reihenfolge der Nachrichten und damit die Konsistenz der Systemtafeln trotzdem gewahrt.

b) Unterschiedliche ISR - Abarbeitungszeiten

Werden beim Bearbeiten der Interrupts Routinen benutzt, die verschiedene zeitliche Längen haben (z.B. wenn ein Prozessor eine Nachricht erhält, der andere aber nicht), so ist durch den unterschiedlichen Bearbeitungsbeginn der darauf folgenden Nachrichten eine korrekte Reihenfolge der Nachrichten bei allen Prozessoren ebenfalls nicht mehr gewährleistet.

Abhilfe für dieses Problem schafft eine einfache Maßnahme: Nach der Registrierung eines Interrupts zur Kommunikation darf während einer Zeitspanne, die zum Entnehmen einer Nachricht aus einem Port ausreicht, kein weiterer Interrupt generiert werden. Vor dem Auslösen eines Interrupts (nicht aber vor dem Übermitteln der Daten) ermittelt deshalb jeder Prozessor, ob diese Mindestzeitspanne T_{ISR} seit dem letzten Interrupt schon vergangen ist.

c) Interrupts bei der Zeitabfrage

Ein Interrupt kann gerade in der Situation erfolgen, in der ein Prozessor vor dem Senden ermittelt hat, daß die oben geforderte Zeitspanne T_{ISR} verstrichen ist. Würde er nach dem Abarbeiten des Interrupts bzw. der ISR an dieser Stelle das Programm wieder aufnehmen, so würde er ohne weitere Zeitabfrage den Interrupt für's Senden sofort (im Gegensatz zu der Forderung nach einer Mindestzeitspanne) auslösen.
Dieses Problem kann dadurch vermieden werden, daß die Instruktionen zwischen der Zeitabfrage und dem Auslösen des Interrupts ununterbrechbar abgearbeitet werden.

d) Verzögerte Interrupts

Führt ein Prozessor gerade eine Instruktion in solch einem nichtunterbrechbaren Codestück aus und tritt ein Interrupt auf, so kann der Prozessor den Interrupt erst nach Ablauf des Codestücks verzögert bearbeiten. Da der Prozessor nach solch einer Zeitabfrage selbst einen Nachrichten-Interrupt generiert haben kann, so bearbeitet er danach die Interrupts gemäß ihren Interrupt-Prioritäten, die anderen aber, die früher angefangen haben, möglicherweise nach der zeitlichen Reihenfolge, was natürlich zu Inkonsistenz in der Nachrichtenreihenfolge führt.

Zur Abhilfe müssen alle Prozessoren vor Abarbeitung des aktuellen Interrupts eine gewisse Zeit zur Synchronisation ihrer ISR abwarten. Da ein Prozessor auch im Betriebssystem für kritische Operationen gegen Interrupts gesperrt sein kann (s. Kapitel 1.3.2), muß für die sperrungsbedingte Wartezeit T_K die maximale Zeit einer solchen Sperrung angenommen werden. Damit ist sichergestellt, daß alle eventuellen Interrupts auch eingetroffen sind. Nach diesem Zeitabschnitt können keine Interrupts mehr von intakten Einheiten eintreffen, da sich alle Prozessoren im "interrupted"-Zustand befinden.

Treffen trotzdem zusätzliche Interrupts ein, so stammen sie von defekten (nichtsynchronisierten) Boards. Um eine Blockierung des Systems mit den Nachrichten defekter SBC (*Life-Lock*) zu vermeiden, kann nach einer solchen Fehlererkennung bei vielen Interrupt-Controllern über ein Maskierungsbit der Interrupt eines als defekt erkannten SBC

wirkungslos gemacht werden.

Das hier vorgestellte Kommunikationsmodell beruht also auf drei Zeitforderungen: einer Synchronisationszeit T_{SYNC} für die Interrupts, einer maximalen Sperrzeit T_K eines Codestücks und einer Mindestzeit T_{ISR} zwischen zwei Kommunikationen. Es ergibt sich also folgendes Bild:

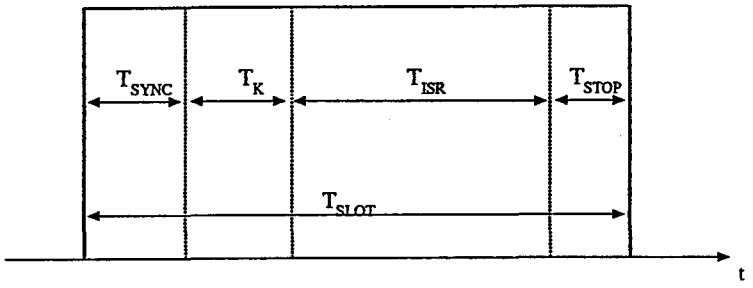


Abb. 2.3.11 Rahmenzeiten des Kommunikationsmodells

Damit wird der Kommunikationsbus zu einem *asynchronen slotted Bus*: Kommunikation ist nur innerhalb von slots (Fächern, Rahmen) zulässig, die innerhalb einer definierten Anfangszeit synchronisiert werden. Wird zusätzlich zu der Wartezeit $T_{SLOT} := T_{SYNC}+T_{ISR}+T_K$ noch ein Sicherheitsabstand T_{STOP} einkalkuliert, so läßt sich das Konzept gut mit dem stark verbreiteten Kommunikationsprotokoll der asynchronen Datenübertragung zwischen Computer und Terminal (V24) vergleichen. Auch hier gibt es eine Synchronisationszeit (Startbit), eine Datenbearbeitungszeit (Datenbits) und einen Sicherheitsabstand (Stopbit). Allerdings unterscheidet sich das vorgestellte Konzept sowohl von dem slotted Bus als auch von der V24-Kommunikation durch die Tatsache, daß der Rahmen eine Norm darstellt, in der ein Nachrichtentransfer von *allen* abgewickelt wird - nicht nur von einem.

Der Prozessor ist nur während der kurzen Synchronisationszeit T_{SYNC} untätig und kann nach der ISR bis zum Ende der Rahmenzeit T_{SLOT} normal arbeiten, so daß keine Prozessorleistung durch das Kommunikationsmodell verloren geht. Wird allerdings ein Prozessor relativ lange (z.B. 14 ms in einer UNIX Implementation) in kritischen Abschnitten vom Betriebssystem gegen Interrupts gesperrt, so wird die durch einen schnellen Bus mögliche, hohe Kommunikationsrate empfindlich begrenzt. Dieser Nachteil kann mit einem eigenen Kommunikationsprozessormodul begegnet werden, wie es auch bei den nachrichtengekoppelten Multicomputersystemen (s. Abschnitt 1.4) üblich ist. Das auf dem Kommunikationsmodul ablaufende Real-Time Betriebssystem ist dabei von den kritischen Abschnitten und den Interrupts der restlichen SBCs befreit, so daß die Kommunikationsrate nur durch die Übertragungsrate und die Nachrichtenverwaltung (Kommunikationsprotokoll) der Kommunikation beschränkt wird.

Flußkontrolle und Fehlertoleranz

Für das Konzept der Nachrichtenübertragung der höhereren Schichten (End-to-End Protokoll der ISO-Schichten, s. Abschnitt 1.4.2) gibt es zwei verschiedene Möglichkeiten: Soll die Kommunikation synchron durch Blockieren des Sendeprozesses, bis die Nachricht zum Empfangsprozeß übertragen wurde, oder asynchron durch Einschalten von Nachrichtenpuffern auf der Sende- und Empfangsseite vorgesehen werden?

Da ein automatisches Blockieren nicht unserem Fehlertoleranzkonzept entspricht, entschieden

wir uns dafür, in ATTEMPTO eine asynchrone Kommunikation zu konzipieren. Dabei wird das folgende logische Protokoll einer asynchronen Nachrichtenübertragung benutzt:

> Der Interrupt aktiviert die Port-Handler aller SBCs. Falls ein SBC zu den Empfängern einer Nachricht gehört, leitet dessen Port-Handler sie an das OS weiter, sendet als ACK-Signal eine Nachricht mit der Signatur der empfangenen Nachricht an deren Absender. Bekommt der Absender keine Empfangsbestätigung (time-out) oder eine solche mit falscher Signatur, so versucht er es mehrmals. Bei Mißerfolg erkennt er auf Bus/ Prozessorfehler und kommuniziert nicht mehr mit der betreffenden Einheit (über diesen Bus).

Durch den Nachrichten-Rückantwortsmechanismus sind zwei Interrupts in kurzem Abstand für den selben Empfänger auf der selben Leitung (vom selben Sender) ausgeschlossen, so daß auch keine Interrupts von intakten Prozessoren verloren gehen können.

Der Sender wartet mit dem Senden der nächsten Nachricht aus seiner Warteschlange solange, bis die Nachricht sicher empfangen (mit ACK quittiert) wurde. Verzögert man den ACK je nach Länge der Warteschlange der empfangenen Nachrichten mit einem Wert, der zwischen Null und (fast) dem time-out Wert des Senders liegt, so läßt sich damit elegant eine für eine asynchrone Datenübermittlung nötige Flußkontrolle implementieren. Für den Fehlertoleranzbetrieb mit seinen kurzen Nachrichtenfolgen reicht dies vollkommen aus, ohne das synchrone Blockieren des Senders durchführen zu müssen. Ist eine explizite Flußkontrolle für die Übertragung großer Datenmengen (Filetransfer usw.) trotzdem nötig, so läßt sich dies durch Protokolle auf höherer Ebene (z.B. XON/XOFF) leicht durchführen.

Ist eine Nachricht empfangen worden, so wird sie in die Warteschlange der empfangenen Nachrichten eingereiht. Um die gleiche Reihenfolge auf allen Prozessoren zu erhalten, muß aber auch beim Empfang einer ungültigen Nachricht (Falsche Signatur) eine Nachricht eingereiht werden. Dies wird durch eine spezielle Nachricht (*Stellvertreter*) erreicht, die nach dem korrekten Empfang überschrieben wird. Erreicht innerhalb eines Zeitfensters (Time-Out!) keine korrekte Nachricht den Empfänger, so wird die Kommunikationsverbindung als gestört betrachtet und die entsprechenden Maßnahmen eingeleitet.

Zusammenfassung

Vergleichen wir die Funktionen dieses Kommunikationssystems mit dem ISO-OSI Modell aus Abschnitt 1.4.2, so implementiert der Kommunikationstreiber die Transportschicht und die tieferen Schichten 2 und 3.

Mit den oben beschriebenen Maßnahmen für Koordination, Fehlertoleranz und Flußkontrolle sind nun die drei verlangten Eigenschaften des *atomic broadcast* verwirklicht: endliche Nachrichtenlänge und endliche Rahmenzeit garantieren endliche Übertragungszeit, Rahmenzeit und Fehlertoleranz implementieren die Atomizität der Nachrichtenübertragung und die Selbstreferenz bei Statusnachrichten bewirkt die identische Nachrichtenreihenfolge bei allen Teilnehmern.

Da unsere Implementation nicht für zeitkritische Anwendungen gedacht ist und die Nachrichten nicht mit fortschreitender Zeit ihre Gültigkeit verlieren können (vgl. Kapitel 3.3), erlaubt der hier beschriebene, in Software ausgeführte Mechanismus zur Datenkonsistenz globaler Systemtafeln im Unterschied zu SIFT [WEN1,2] und FTMP [HOP] den Verzicht auf eine globale Systemuhr mit allen ihren Problemen [PEA].

2.3.5 Die Parallelisierungskonzepte von ATTEMPTO

Die beiden Systeme ATTEMPTO 1 und ATTEMPTO 2 unterscheiden sich in ihren Parallelisierungsmechanismen stark voneinander. In ATTEMPTO 1 besteht die zu verteilende Softwareeinheit aus einem Job, das heißt, aus einem kompletten Programm. Dies ist zwar einfach zu realisieren, setzt aber als Belastung des Betriebssystems die Existenz vieler kurzer Programme voraus, so daß nur durch die Verteilung der Jobs das Gesamtsystem gut ausgelastet werden kann. Besteht die Last aber im Wesentlichem aus einem einzigen Programm, so ist keine Lastverteilung möglich und das Multiprozessorsystem hat insgesamt einen niedrigen Wirkungsgrad. Um auch hier eine Lastverteilung erreichen zu können, muß eine feinere Granularität (s. Abschnitt 1.3.2) der verteilbaren Programmstücke gewählt werden. Dies ist im Konzept von ATTEMPTO 2 berücksichtigt.

ATTEMPTO 1: Das Dispatching der Benutzerprogramme

Hat der Benutzer spezifiziert, welches Programm und, implizit mit dem Fehlertoleranzgrad, wie viele Kopien von dem Programm von verschiedenen Prozessoren ausgeführt werden sollen, so muß nun der Auftrag koordiniert ausgeführt werden.
Das Dispatching wird nicht vorher deterministisch festgelegt wie beim Pre-Scheduling bei [WEN2] und [FÄR], sondern nach dem Prinzip der Anziehung (*attraction-principle*) während der Laufzeit durchgeführt. Im Gegensatz zu der zentralen Hardware-Version dieses Prinzips in PLURIBUS [KAT] konzipierten wir eine dezentrale Software-Version:

> Jeder freie Prozessor bewirbt sich um das in seiner Systemtafel als 'noch zu bearbeiten' gekennzeichnete, in der Eingabereihenfolge nächste Benutzerprogramm. Empfängt er eine Bewerbung für ein solches Programm, so trägt er den Bewerber dafür ein. Ist er selbst der Bewerber, so beginnt er mit der Ausführung des Programms.
> Alle Bewerbungen für bereits "besetzte" Programme werden ignoriert, außer der eigenen: Diese löst stattdessen eine Bewerbung um das nächste freie Programm aus.

Dieses Prinzip ist in dem Fehlertoleranz-Dispatcher Modul FTD verwirklicht (s. Abb. 2.3.12). In der nachfolgenden Abbildung ist das aktive Modul in seinen Funktionen kurz charakterisiert.

MODULE FTD	
Datenstruktur:	Job Control Buffer JCB : Liste von Job-Kontroll-Blöcken mit JobNamen, Fehlertoleranzindex, Kollegen-Liste, Anfangszeiten (für Timeout)
aktive Einheit:	Fault-Tolerance Dispatcher, verwaltet den JCB-Puffer
Botschaften:	
newJob:	Erzeugung eines Job-Kontroll-Blockes, falls idle, Initiierung der Job-Bearbeitung, Mitteilung dieser Intention allen SBCs (*'StartRequest'*)
StartRequest:	falls noch nicht alle Kollegen vorhanden, Eintragen der Start-Zeit in zugehörigen Job-Kontroll-Block, und falls zudem eigene StartRequest-Botschaft, Botschaft an PostOffice für JobStart
endJob:	Terminierung einer Job-Bearbeitung: Löschen des JCB Eintrags sowie Botschaften *'clearDIB'* an DIBclerk, *'clearDOB'* an DOBclerk, *'clearSAB'* an SABclerk

Abb. 2.3.12 Der Fehlertoleranz-Dispatcher

ATTEMPTO 2: Das Parallelisierungskonzept

In der Nachfolgeversion von ATTEMPTO wurde ein Schema für eine Parallelarbeit mit feinerer Granularität (s.Abschnitt 1.3), aber auch komplizierterem Ablauf konzipiert.

Ausgehend von der Überlegung, daß es sinnvoller ist, die dem Problem inhärente Parallelität *explizit* im Programm auszudrücken als dies einem "intelligentem"(?) Compiler zu überlassen, wählten wir als Parallelisierungsebene die *Spezifikation* im Programm durch das PARBEGIN/ PAREND-Konstrukt (s. Abschnitt 1.3.1). Die Implementierung in dem nachrichtenorientierten ATTEMPTO 1 - System durch *send()* und *receive()* Aufrufe sollte dabei dem Programmierer verborgen bleiben.
Dies läßt sich in folgenden Konzept-Punkten beschreiben (s. [LUT1]):

* Der Programmierer spezifiziert in seinem Programm nur die Menge der parallel ausführbaren Aktionen; die Konfiguration und der Mechanismus der Verteilung auf die vorhandenen Prozessoren bleiben verborgen.
* Das Programm soll unabhängig von der Zahl der bei der Ausführung beteiligten Prozessoren das selbe Ergebnis bringen. Damit sind Umkonfigurierung des Systems (Ausfälle!) und Portierung des Programms leicht möglich.
* Während der Compilierung sollte eine möglichst weitgehende, maschinelle Überprüfung der konsistenten Nutzung von parallelen Konstrukten im Programm durchgeführt und die nachrichtenorientierte Implementierung durch *send()* und *receive()* Aufrufe automatisch generiert werden.
* Der Aufruf und die Parameterübergabe von parallel ablaufenden Aktionen sollte automatisch generiert werden. Auch die Eingliederung erhaltener Ergebnisse erfolgt automatisch.

Dazu sollen folgende Sprachkonventionen verwendet werden:

- *PARBEGIN und PAREND*
  Alle parallel ausführbaren Aktionen sind zwischen PARBEGIN und PAREND aufgeführt. Damit ist PAREND als Barriere (s. Abschnitt 1.3.1) der Punkt, an dem die parallelen Aktionen terminieren und die Ergebnisse der Aktionen verarbeitet werden müssen.
- *Mittlere Granularität*
  Die mit PARBEGIN und PAREND geklammerten, parallel ausführbaren Aktionen (*paralleler Bereich*) sind keine Elementarbefehle (feine Granularität), sondern ganze Befehlsabschnitte (mittlere Granularität). Dies ist besonders bei nachrichtenorientierten Systemen sinnvoll, da der Mehraufwand für Aufruf und Parameterübergabe von parallel ablaufenden Aktionen meist mehrere Elementarbefehle ausmacht.
  Der sequentielle Befehlsabschnitt kann mit extra Klammern (z.B. *PB* und *PE*) gekennzeichnet werden, oder aber durch die *Einschränkung der parallelen Aktionen auf Prozeduren.*
- *Makro-Anweisungen*
  Werden viele, gleichartige, aber mit unterschiedlichen Parametern aufgerufene Aktionen in einem parallelen Bereich verwendet, so läßt sich dies mit *Makro-Anweisungen* (z.B. FOR i:=1 TO 10 DO) spezifizieren, die dann zur Compile-Zeit zu Anweisungen für echt parallele Aktionen expandiert werden.

Auf die Einführung eines *pipe-Konstrukts* (z.B. *PIPEBEGIN / PIPEEND*) für Pipeline-ähnliche Parallelität kann man zunächst verzichten, da bereits mit PARBEGIN/PAREND eine Pipeline installiert werden kann. In der folgenden Abbildung ist ein Beispiel einer zweistufigen Pipeline gegeben, bei dem das Hauptprogramm die Rolle der Arbeitsvermittlung (Übertragung) zwischen der Prozedur p1 und der Prozedur p2 einnimmt.

```
Y:= Null;
LOOP
Input(X);                (* Neues X *)
   PARBEGIN
       p1 (VAR X);       (* als Eingabe der ersten Stufe *)
       p2 (VAR Y);
    PAREND               (* Alter Wert von X wird überschrieben *)
Output(Y);  Y:=X;        (* und zur Eingabe Y der nächsten Stufe *)
ENDLOOP
```

Abb. 2.3.13 Pipeline Konstruktion mit PARBEGIN/PAREND

Das Implementierungskonzept

Die Umsetzung der Spezifikation in tatsächlich parallel ablaufende Aktionen läßt sich auf verschieden Wegen erreichen. Der vollständigste Ansätz dürfte sicher in der Erstellung eines eigenständigen, neuen Compilers einer neuen, parallelen Sprache liegen. Dem stehen allerdings Probleme wie Portabilität, Arbeitsaufwand und Stabilität einer Neuimplementierung gegenüber.

Stattdessen entschieden wir uns für die Idee, die Prozeduren eines parallelen Bereichs als "remote procedure calls" (RPC, s. Abschnitt 1.3.3) aufzurufen. Ordnet man jedem RPC einen Leichtgewichtsprozeß zu, so realisieren die unabhängig voneinander blockierenden Leichtgewichtsprozesse parallel ablaufende "remote service invocations" (RSI) des Hauptprogramms [KAM].

Dieses Konzept ermöglicht uns, anstelle einer neuen Programmiersprache eine bestehende, blockorientierte Sprache (Modula-2) mit parallelen Sprachkonstrukten dadurch zu erweitern, daß man die Schlüsselworte von einem Präprozessor im Programmtext durch Laufzeitprozeduren einer besonderen Bibliothek ersetzen läßt. Dies bedeutet insbesondere (s. [LUT2]):

- PARBEGIN und PAREND werden in Laufzeitprozeduren ParBegin(..) und ParEnd(..) umgesetzt. Diese Prozeduren aktualisieren die interne Nummer des parallelen Bereichs und stoßen eine Prozedur an, die auf alle Ergebnisse wartet und sie dann eingliedert.
- Für jede in einem parallelen Bereich aufgerufene Prozedur generiert der Präprozessor eine Stub-Prozedur (s. Abschnitt 1.3.3), die die Zwischenschicht zwischen Prozeduraufruf und Nachrichtenaustausch realisiert.
- Beim Aufruf einer Prozedur im parallelen Bereich wird bei der Übergabe eines Referenzparameters (VAR-Parameter in Modula-2) nicht nur die Parameteradresse, sondern auch der Parameterwert sowie seine Speichergröße (SIZE) übergeben. Adresse und Größe werden von ParEnd() ausgewertet.
- Kopien des Hauptprogramms mit allen Prozeduren werden auf den beteiligten Prozessoren gehalten. Die Unterscheidung in Hauptprogramm oder *Auftraggeber* (AG) und aufgerufene Prozedur-Service oder *Auftragnehmer* (AN) wird nach der Prozeßverteilung bei der Initialisierung vorgenommen.
- In den RPC-Prozeduren dürfen keine globalen Variablen verwendet werden.

Die Überprüfung des Gebrauchs globaler Variable läßt sich modularisieren, wobei die Module um Pseudokommentare erweitert werden [LUT2]. Der falsche Gebrauch von globalen Variablen läßt sich allerdings nicht völlig vermeiden; hier müßte sonst der Compiler mit allen verfügbaren Informationen umgeschrieben und erweitert werden.

Die Verwendung einer einheitlichen, nachrichten-orientierten Kommunikationsschicht auf Hochsprachenebene ermöglichte uns, das existierende LAN und Multiprozessorsystem in der Laufzeitbibliothek einheitlich anzusprechen [KAM].

Ist das Zielsystem ein homogenes, aus gleichartigen Prozessoren bestehendes Multi-Mikroprozessorsystem wie ATTEMPTO 1, so kann der RPC sehr schnell und effektiv durch das Verschicken des Prozeduraufruf-Stacks in den Auftragsnachrichten implementiert werden [LUT2].

Die Lastverteilung der RSI auf die beteiligten Prozessoren sollte sowohl vom Kommunikationsaufwand (Zahl und Größe der Parameter, etc) als auch von der Laufzeit-Länge der parallelen Aktionen abhängen. Um für einen guten Schedule Anhaltspunkte zu gewinnen, wird das Programm zunächst auf einem einzigen Prozessor gestartet und ein Ausführungsprofil erstellt. Mit Hilfe dieser ersten Schätzung und den Informationen über die parallele Struktur, die der Präprozessor erstellt hat, kann nun ein Scheduler eine Tabelle erstellen, die zur Laufzeit eine statische, im Mittelwert optimale Verteilung bewirkt. Dabei kann dieser Schedule nur eine suboptimale Näherung sein, da das Problem NP-vollständig ist. Eine geeignete, suboptimale Scheduling-Strategie zu finden ist allerdings noch Gegenstand der Forschung.

2.3.6 Simulation und Emulation des Systems

In diesem Abschnitt soll ein Ansatz gezeigt werden, der mit relativ einfachen Mitteln die Simulation des Multiprozessorsystems als Multi-Prozeß-System durchführt und dabei die Software-Module des Fehlertoleranzsystems verwendet [BR7]. Dadurch wird das ATTEMPTO System nicht nur simuliert, sondern teilweise auch emuliert.
Insbesondere wird eine Abbildung der Soft- und Hardware-Charakteristika des Multi-Mikroprozessorsystems auf die unter UNIX gegebenen System-Möglichkeiten durchgeführt sowie die Maßnahmen zum Testen, Validieren und Monitoring einer derart komplexen Software beschrieben.

Diese Simulation ermöglicht darüber hinaus auch die Tolerierung transienter Fehler während der Simulation. Im Unterschied zum ATTEMPTO System wird dazu keine Hardware-Redundanz (multiple SBC) genutzt, sondern die Zeitredundanz der simulierten SBC-Prozesse (vgl. Abschnitt 1.4.1). Mit der Simulation ist es auch möglich, andere modellierte Diagnoseverfahren zu implementieren, sie auf Konsistenz zu testen sowie ihren Kommunikationsaufwand zu messen. Auch eine Erweiterung des ATTEMPTO-Systems zur verteilten Zusammenarbeit der SBC (pipelining etc.) ist damit leicht testbar.

Die Prozeßstruktur der Simulation

Die Simulation hat die Aufgabe, die vorgegebe Hard- und Softwarestruktur zum Testen und Validieren der logischen Grundfunktionen (Interprozessor-Kommunikation und Koordination) sowie der Fehlertoleranzfunktionen nachzubilden.
Dies bedeutet u.a.

- Die Prozeßstruktur des simulierten Systems sollte Untermenge der Simulation sein
- Als Grundmechanismen der Kommunikation sollten die Hardware-Mechanismen des Zielrechners logisch emuliert werden.

Dazu wird das Prozeßsystem, wie es in Abschnitt 2.3.3 beschrieben und in Abb. 2.3.3 abgebildet ist, in mehrfacher Ausführung auf einem Rechner erzeugt. Betrachten wir dazu für das Beispiel von drei SBC die Abbildung 2.3.14 . Zum Aufbau des Simulationsrahmens wird ein Initialisierungsprozeß (gestrichelte Linien) gestartet. Dieser erzeugt alle FTL-Prozesse sowie einen Supervisor-Prozeß (SV). Der SV verfügt über ein System überlappender Fenster (windows), die es gestatten, sowohl die Ausgabe auf das simulierte Terminal als auch Kontrollnachrichten einzelner FTL übersichtlich getrennt darzustellen. Außerdem wird damit ermöglicht, Benutzereingaben (TTY) an alle SBC oder für Kontroll- und Monitorzwecke an einzelne, selektierte SBC zu geben.
Die Fehlertoleranzschicht FTL sowie die Prozesse TT, PH, SH und UJ sind pro SBC einmal vorhanden und mit den gleichen Kommunikations-pipes versehen wie in ATTEMPTO.

Für die Simulation wurden die Anforderungen an den Benutzerjob zunächst geringfügig modifiziert: Der Job darf kein eigenes Terminal öffnen und keine Kinder erzeugen. Mit dieser Einschränkung ist es nicht mehr nötig, die SysCalls des Benutzerjobs UJ zu überwachen. Stattdessen werden der shell (und damit auch dem UJ) pipes für Standardinput und Standardoutput zur Verfügung gestellt. Beim exit()-SysCall des UJ wird automatisch der shell die Kontrolle übergeben, die mit einer speziellen Ausgabe vor ihrem exit() dem SC Prozeß das Ende des Benutzerjobs anzeigt. Dies wird von SC in eine entsprechende Nachricht an die FTL umgesetzt.

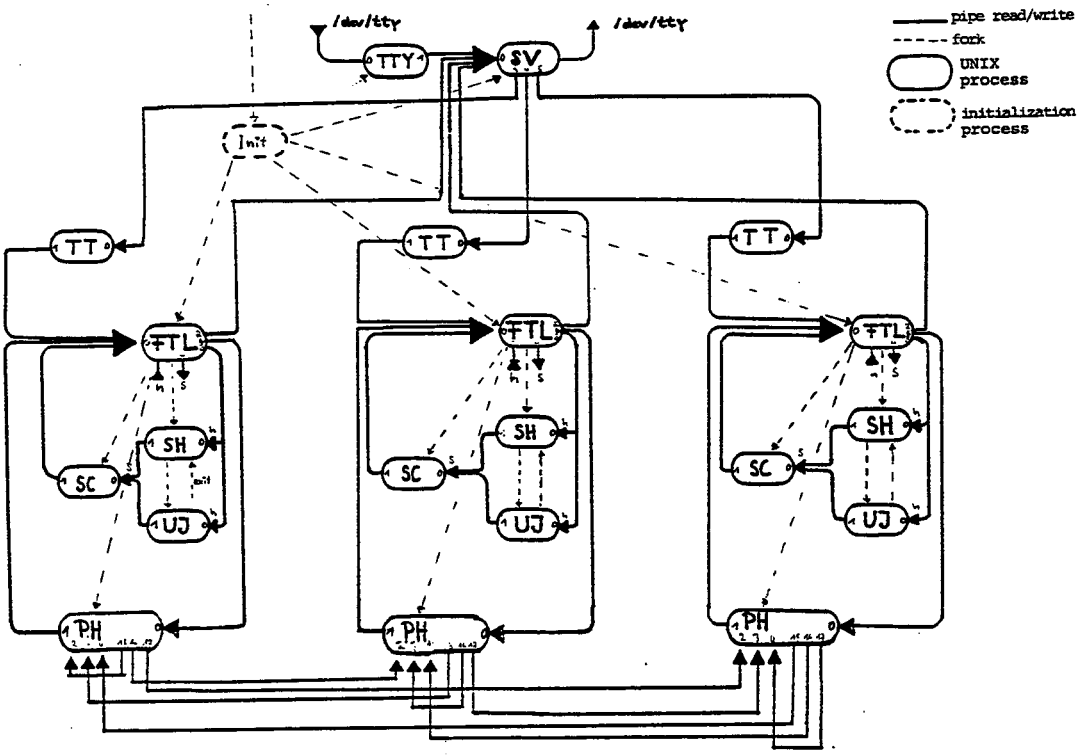


Abb. 2.3.14 Simulation der UNIX-Prozeßkonfiguration

Für die Interprozessor-Kommunikation werden die Kommunikationskanäle der obersten OSI-Schichten (s. Abschnitt 1.4.2) durch pipes nachgebildet und die Sendeseite der Porthandler durch einen eigenständigen UNIX-Prozeß PH. Die Interrupts, die auf den SBC das Auslesen der ports bewirken, lassen sich durch Signale des UNIX-Systems ersetzen. Dabei wird von der Möglichkeit des UNIX-Systems Gebrauch gemacht, ein Signal an alle Prozesse eines Terminals zu senden (Broadcast). Um die nicht beteiligten Prozesse damit nicht abzubrechen, werden alle Prozesse (bis auf die PH) bei der Erzeugung unempfindlich gegenüber diesen Signalen gesetzt.

2.3.7 Implementierung des Systems

Das ATTEMPTO 1 System wurde nach der Konzipierungs-und Design-Phase ab 1981 und einer Kodierungsphase ab 1982 zuerst im Rahmen einer Software-Simulation und Emulation (s. Abschnitt 2.3.6) auf einer PDP-11 vom Autor installiert. Bei der weiteren Entwicklung des Projekts stellte sich heraus, daß es besser ist, anstelle Unix Version 6 und 7 kompatible Betriebssystemkerne zu verwenden, auf einer neueren System V Unix-Portierung für SingleBoard Computer aufzusetzen.

Hardware

Da auch ein Massenspeicher für das Auslagern von Prozessen (swapping) bei stark gestiegenen Umfang des Unix-Kerns nötig war, zeigte sich bald die Notwendigkeit, von der ursprünglichen Hardware-Implementierung auf drei Intel iSBC 86/12a Single-Board Computer Platinen mit 8086 Prozessor und 32kB Dual-port Hauptspeicher abzugehen und ein komplettes Hardware-Redesign vorzunehmen.

Ab 1986 wurde nun ein neues Zielsystem ausgesucht und beschafft. Es besteht aus drei SBC-Platinen Heurikon HK68/V10 mit je einem Motorola 68010 Prozessor, 1MB Dual-port Hauptspeicher, SCSI-Schnittstelle und seriellem I/O. Außerdem wurde an jedem SBC als lokale Erweiterung des Hauptspeichers eine Festplatte ($5^{1/4}$", 150MB) angeschlossen. In der Abbildung 2.3.15 ist ein Foto der Anlage abgebildet.

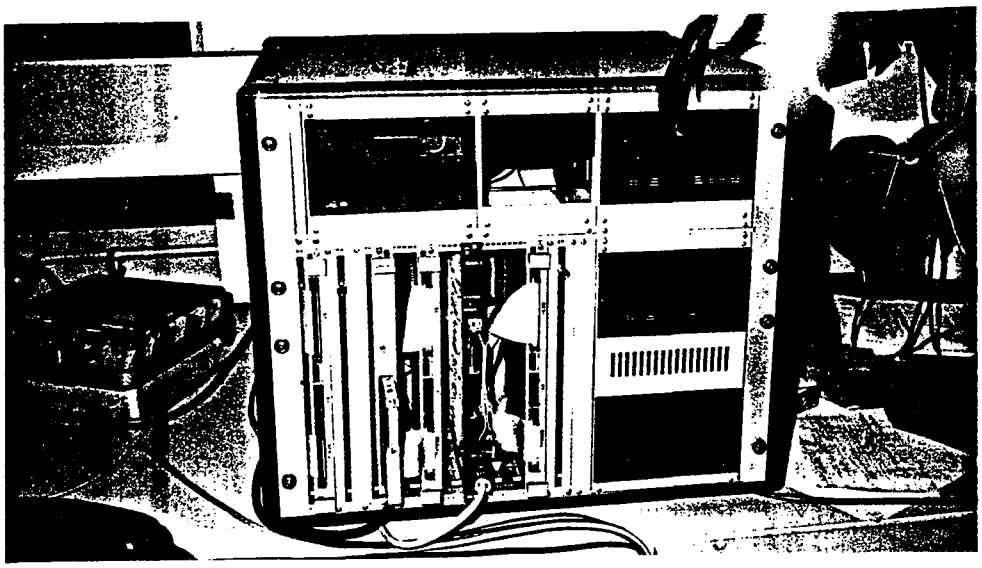


Abb. 2.3.15 Das aktuelle ATTEMPTO 1 System

Im Bild sind die drei Prozessorplatinen erkennbar, an deren SCSI- und RS232-Ausgängen die entsprechenden Kabel montiert sind. Zusätzlich sind noch ein Hardware-Busmonitor und ein Ethernet-Controller als Einschübe vorhanden.
Man sieht, daß die physikalischen Ausmaße stark durch den Formfaktor der $5^{1/4}$" Festplattenlaufwerke bestimmt werden. Dies läßt sich bei einer späteren Umrüstung auf $3^{1/2}$" Laufwerke korrigieren.

Die Spannungsversorgung

Die drei Single-Board-Computer sind als Einschubkarten an dem Multi-Master Bus *VME-Bus* angeschlossen, der als *Backplane* elektrisch bezüglich der Spannungsversorgung in drei Segmente geteilt ist. Obwohl die Möglichkeit besteht, jedes Segment einzeln mit einem Netzteil und einem Prozessor seperat zu betreiben, entschlossen wir uns doch dazu, die drei Netzteile über Dioden entkoppelt an die gemeinsamen Spannungversorgungsleitungen zu schalten. Diese Lösung schützt zwar nicht vor den seltenen Überspannungsfehlern; gegen diese existiert intern in jedem Netzgerät eine Schutzschaltung. Dafür aber ermöglicht es, die häufigeren Ausfälle der Spannungversorgung zu tolerieren.

Abgesehen von den für fehlertoleranten Betrieb ausgelegten Netzteilen gibt es noch verschiedene Einheiten im System, die einzig sind und deshalb konzeptionell als *Single-Point-of-Failure* angesehen werden müssen. Im Fehlertoleranzsinn "kritische" Komponenten sind der Terminal-Bus (s.u.) inklusive Terminal, die LAN-Verbindung (s.u) und die globalen Plattenpartitionen.

Zwar ist der VME-Bus im implementierten System auch nur einmal vorhanden, er kann aber als Kommunikationsweg prinzipiell mehrfach vorhanden sein (Mehrbussystem!) und transparent für das Benutzerprogramm und die Fehlertoleranz-Zwischenschicht von den unteren Kommunikationsschichten (s. Abschnitt 1.4.2) verwaltet werden; selbst die Auswahl zwischen alternativen Kommunikationswegen (z.B. VMS-Bus) zur Performance-Steigerung und Fehlertoleranz kann transparent durchgeführt werden. Damit ist der Systembus ein aktueller, aber kein prinzipieller "Single-Point-of-Failure".

Terminal I/O

Die Verbindung zum (einzigen!) Terminal des Benutzers geschieht über die Terminalleitung, eine abgeschirmte V24-Leitung. Alle Eingänge der SBC sind direkt miteinander verbunden, so daß ein Signal vom Terminal alle SBCs direkt erreicht. Da ein Zeichenecho vom Computer (Vollduplex-Betrieb) nicht so einfach möglich ist (Welcher SBC soll es machen?), wird das Bildschirmecho im Terminal am Anfang lokal erzeugt. Jeder SBC, der etwas ausgeben will, bewirbt sich um die Terminalleitung als globale Resource mit dem üblichen Protokoll (s. Abschnitt 2.3.3) und kann dann Zeichen zum Terminal senden, ohne daß der Zeichenstrom durch andere Prozessoren unterbrochen wird. Dies läßt sich in einer komfortableren Version dazu einsetzen, daß jedem Zeichenstrom eine Kennung vorausgestellt wird, die nicht vom Benutzer erkannt ("Job xyz gibt aus.."), sondern von einem "intelligenten", fensterorientierten Terminal (in unserem Fall: ein ATARI ST) dazu benutzt wird, den Ausgabestrom auf eine bestimmte Fensterfläche zu lenken. Damit besteht auch die Möglichkeit, bei interaktiven Programmen (s. Abschnitt 2.3.3) das lokale Echo abzuschalten und unter ausschließlicher Kontrolle eines SBC "wie auf einem normalen Computer" zu arbeiten. Das Fensterprogramm kann zusätzlich bei Eingaben des Benutzers die Fensterkennung der Eingabe voranstellen, so daß jeder SBC weiß, ob die Eingabe für ihn bestimmt ist.

Die Fehlerkontrolle bei der Ausgabe (s. Abschnitt 3.4.3) wird durch eine Rückführung des Ausgabesignals in den Eingang erreicht. Eine einfache elektrische Kopplung (s. Abb. 2.3.16) erlaubt es allen SBC, den Ausgang "mitzuhören".

Durch einen Unterschied im Datenformat zwischen Terminal-Sender und SBC-Sender (Bit 8 ist ON bzw.OFF) läßt sich leicht herausfinden, ob ein Zeichen vom SBC ausgegeben oder vom Benutzer eingegeben worden ist.

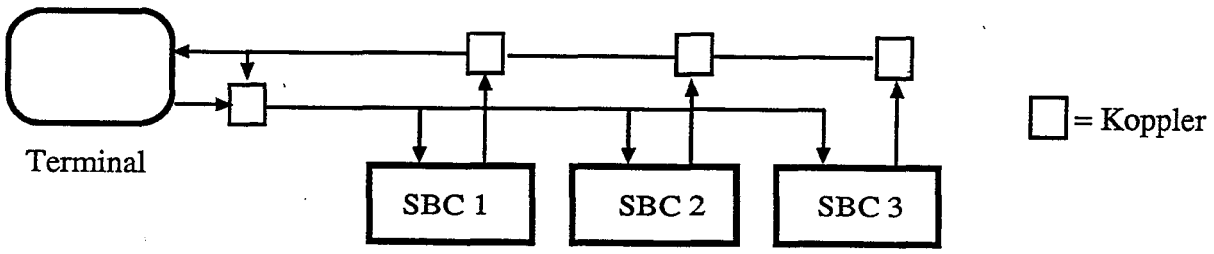


Abb. 2.3.16 Terminal-Bus Schaltung

Netzwerkanbindung

Obwohl ATTEMPTO als Einbenutzersystem konzipiert ist, ist für den Datenaustausch (Electronic mail etc) eine Netzwerkanbindung wünschenswert. Wie kann sie in das Fehlertoleranzkonzept integriert werden?

Obwohl es für die Fehlertoleranzmechanismen einfacher wäre, mit drei unabhängigen Netzwerk-Controller als unabhängigen Eingabemodulen analog zu dem Terminal I/O zu arbeiten, gliederten wir in das implementierte System nur einen einzigen Netzwerk-Controller ein, da ein Benutzer in dieser Hinsicht auf Kostengründen im Normalfall auf Redundanz verzichten wird.

Obwohl damit eine globale Resource geschaffen wird, sollte doch nach dem Fehlertoleranzkonzept (s. Abschnitt 3.4.1) der Datenaustausch unabhängig von einem einzelnen Prozessor erfolgen. Da der Verbindungsaufbau im Netzwerk (die Initialisierung des Controllers) aber nur einmal erfolgen kann, muß dies durch einen Prozessor entweder *vor* einem Fehlertoleranzbetrieb oder im Fehlertoleranzbetrieb nach einem Abstimmungsprotokoll durch einen IO-Master geschehen. Es bietet sich deshalb an, das Abstimmungsprotokoll von der Ausgabe abzulösen und generell für die Bedienung aller globalen Resourcen einzusetzen. Damit kann auch die Erzeugung und Vernichtung von globalen Resourcen, wie sie die Kommunikationsendpunkte (*sockets*) darstellen, fehlertolerant kontrolliert werden.

Bei der Integration des Netzwerk-Controllers gibt es zwei Design-Alternativen: die Integration des Netzwerk-Controllers als aktive Einheit, die sich am Nachrichtenverkehr (atomic broadcast) beteiligt, und der Anbindung als passive, Datenpuffer enthaltende Einheit.

Die aktive Lösung war in unserer Implementierung nicht möglich, da der Quellcode der Kommunikationssoftware auf dem Netzwerkcontroller nicht zur Verfügung stand und das atomic broadcast Protokoll deshalb dort nicht eingebunden werden konte. Ein zweites Problem stellt das Hardwareprotokoll auf dem VME-Bus dar, das nach dem Auslösen eines Interrupts durch den Netzwerk-Controller ("Daten erhalten") mit einem einzigen ACK beendet werden muß. Auch hier verbietet sich wiederum eine Änderung des Protokolls, da der Quelltext fehlt.

Stattdessen integrierten wir den Netzwerk-Controller als passive Einheit. Wurde ein Datenpaket empfangen, so benachrichtigt der Netzwerk-Controller alle SBCs durch einen Interrupt von dem Ereignis. Jeder SBC, der auf Daten aus dem Netz wartet, liest spezielle Controller-Register aus und erfährt dort, für welchen TCP/IP-Port (s. Abschnitt 1.4.2) die Daten bestimmt sind und wo im Dual-Port Speicher des Controllers sie stehen. Der erste SBC, der sie liest, setzt außerdem in einer test-and-set Operation eine Semaphore und führt den Hardware- ACK aus [GÜN1].

Damit ist die Dateneingabe, ähnlich wie beim Terminal, wieder prozessorunabhängig; der Netzwerk-Controller läßt sich mit dem UART der seriellen Eingabe vergleichen, das ebenfalls einen Interrupt und einen Datenpuffer zur Verfügung stellt.

Interprozessorkommunikation

Die Interprozessorkommunikation konnte ausschließlich mit den Standardkomponenten (s. Vorgabe in 2.3.1!) des SBC realisiert werden. Betrachten wir dazu das Blockschaltbild des SBC in Abbildung 2.3.17.

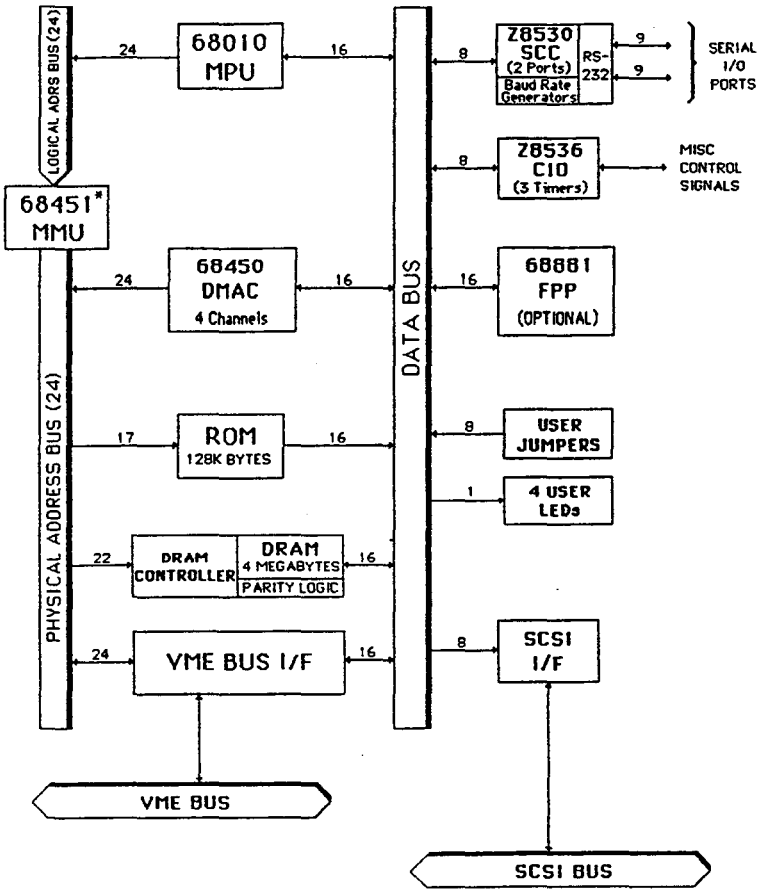


Abb. 2.3.17 Blockschaltbild eines Single-Board-Computers

Wie man sehen kann, gibt es verschiedene Möglichkeiten, die in Abschnitt 2.3.4 entwickelten Hardwarekonzepte zu realisieren. Durch eine geeignete Programmierung der PALs wurde der globale VME-Bus Adressraum in "ports" fehlertolerant (s. Abschnitt 3.4.3) aufgespalten. Die für einen "atomic broadcast" nötige, gleiche zeitliche Reihenfolge der Nachrichten auf allen SBCs (s.Abschnitt 2.3.4) wurde durch die Assignierung der Interruptleitungen des VME-Bus zu den Eingängen des Interruptregister-Bausteins jedes SBCs unterstützt [GÜN1]. Tritt ein Interrupt auf, so wird das entsprechende Bit im Register gesetzt, falls es nicht maskiert wurde. Damit wird zwar nicht die echte zeitliche Reihenfolge der Interrupts gespeichert, aber durch die Bitstelle im Register eine feste Reihenfolge beim späteren Auslesen induziert gemäß dem im Abschnitt 2.3.4 entwickelten Konzept.

Da mit dem vorliegenden Konzept der Nachrichtenkopplung der Prozessoren jede Nachricht *vier Mal* kopiert werden muß, bevor sie beim Empfängerprozeß ankommt (vom Sendeprozeß zum

Porthandler-Puffer, Sender PH-Puffer zum Empfänger-Port, Empfänger-Port zum PH-Puffer, PH-Puffer zum Empfänger-Prozeß), lohnt es sich, zum schnelleren Kopieren den vorhandenen DMA-Chip in einer besonderen low-level Prozedur (Assembler) einzusetzen.

Software

Die für den Power-up Selbsttest verwendeten Prozeduren wurden in einem Boot-EEPROM untergebracht; die Prozeduren der transparenten Selbsttests dagegen (s. Abschnitt 3.4.2) setzen eine überwiegende Korrektheit des Prozessors (s. Abschnitt 1.5) voraus; sie wurden als ein Pseudo-device an den UNIX-Kern angebunden. Damit sind sie einerseits leicht modifizierbar und trotzdem nur im geschützten Systemmodus mit klarer Schnittstelle aufrufbar.

Das gesamte Softwaresystem wurde so zurechtgeschnitten, das die für den kleinen Hauptspeicher der PDP-11 (letzte Ausbaustufe: 256kB!) nötigen Overlays der Software transparent vom Dispatcher der mittels Modula-2 Coroutinen [WIR] realisierten Leichtgewichtsprozesse verwaltet wurden. Der Vorteil dieser Methode zeigte sich bei der Portierung auf das aktuelle ATTEMPTO 1-System: Die Umstellung auf NON-Overlay Betrieb machte keine größeren Programmänderungen nötig.

Da die send() und receive()-Konstrukte sowie die Prozeduren zum Öffnen und Schließen der Interprozess-Kommunikationskanäle durch eine Hochsprachen-Schnittstelle spezifiziert wurden, konnten Präprozessor und die Laufzeitumgebung der Parallelisierungsumgebung von ATTEMPTO 2 zunächst auf dem LAN-Rechnersystem des Lehrstuhls implementiert werden [KAM]. Damit ist die Portierung auf das zur gleichen Zeit in der Implementierung befindliche ATTEMPTO 1-System mit einem sehr geringen Aufwand an Modifikationen möglich.

3.0 Fehlertoleranz in Multi-Mikroprozessorsystemen

Die zur Erzielung von Fehlertoleranz notwendige Redunz kostet, wie wir in Abschnitt 1.4 gesehen haben, immer Mehraufwand. Standard-Fehlertoleranz-Verfahren, wie EDC (Error Detection and Correction) und CRC (Cyclic Redundancy Check), die auf Daten- und Zeitredundanz beruhen (s. [GÖR]), finden zur Absicherung zuverlässigkeitskritischer Systemkomponenten in Rechensystemen zunehmend Verwendung, da sie sich verhältnismäßig kostengünstig in VLSI-Technologie realisieren lassen. Bei Einsatz von massiver Redundanz - z.B. durch Vervielfachung ganzer Rechnereinheiten - muß jedoch mit etwa drei- bis sechsfach gestiegenem Kostenaufwand im Vergleich zu einem nicht fehlertoleranten Einzelrechner gerechnet werden. Es ist daher immer zu prüfen, ob der erreichte Zuverlässigkeitsgewinn diese erhöhten Aufwendungen rechtfertigt. Hierbei ist der Begriff "Zuverlässigkeit" als qualitative Systemeigenschaft gebraucht, wie es auch im allgemeinen deutschen Sprachgebrauch üblich ist. Zuverlässigkeit hat jedoch unterschiedliche quantitative Aspekte, die die anwendungsbezogenen Anforderungen an ein System präziser beschreiben. Man unterscheidet :

- *hochzuverlässige Systeme:*

  Konstruktionsziel ist hier das Überleben für einen bestimmten Einsatzzeitraum.

  Die geeignete Methode zur Realisierung solcher Systeme ist Fehlermaskierung, wenn gleichzeitig strengen Echtzeitanforderungen Rechnung getragen werden muß und lange Lebensdauer nicht gefordert ist. Von Vorteil für die Echtzeitverarbeitung ist, daß keine Maßnahmen zur Fehlerdiagnose und Rekonfiguration wie bei Verwendung von dynamischer Redundanz nötig sind, was zugleich die Implementierung vereinfacht und zu hoher Fehlerabdeckung führt. Demgegenüber steht der hohe Kostenaufwand (wenigstens Verdreifachung erforderlich) und daß wegen der fehlenden Ausgrenzung bzw. Ersetzung von Defektkomponenten die Einsatzdauer begrenzt ist. Deshalb kommen für bestimmte Anwendungen gelegentlich Mischformen der Redundanz zum Einsatz (Hybridredundanz).

- *hochverfügbare Systeme.*

  Konstruktionsziel für diese Systeme ist, die nutzbare Systemzeit so groß wie möglich zu machen. Dies läßt sich nicht nur durch lange Überlebenszeiten wie bei den hochzuverlässigen Systemen realisieren, sondern auch durch kurze Diagnose, Rekonfigurations- und Reparaturzeiten.

  Als Beispiel für hochverfügbare Systeme eignen sich Fehlertoleranzkonzepte für die Steuerung elektronischer Vermittlungsanlagen. In diesem Anwendungsfeld existieren traditionell sehr hohe Anforderungen an die Verfügbarkeit: weniger als 3 Minuten Ausfallzeit/Jahr bei einer Gesamtlebensdauererwartung von 30 bis 40 Jahren. Dies entspricht praktisch ununterbrechbarem Betrieb, jedoch bei manueller Reparierbarkeit. Demgegenüber sind die Forderungen nach Integrität gering: falsche oder zwangsausgelöste Verbindungen sind bis zu einem gewissen Grad tragbar. Eine weitere wichtige Forderung, die gleichzeitig erfüllt sein muß, betrifft die modulare Erweiterbarkeit solcher Anlagen, was sich auf die Architektur der Steuerung, die meist funktionsbezogen hierarchisch konzipiert ist, auswirkt.
  Ein geeigneter Ansatz für Fehlertoleranz ist dynamische Redundanz (Verdoppelung) etwa im Duplexbetrieb. Beispiele sind SSP113 (Siemens EWSD), Bell 3B20D und andere.

- *sichere (integre) Systeme.*

  Hier kommt es in erster Linie auf die Verhinderung gefährlicher Zustände an (*Fail Save*-Eigenschaft, s. Abschnitt 1.4). Integre Rechensysteme benötigen zwar Redundanz zur Fehlererkennung, um im Notfall gefährliche Ausgaben zu verhindern, sind jedoch nicht eigentlich fehlertolerant, da sie eine Fortführung des Betriebs nicht gewährleisten. Anwendung finden solche Systeme etwa bei der Steuerung von Signalanlagen (sicherer Zustand: Halt) oder Reaktorsteuerungen (Notstopp); ein geeignetes Fehlertoleranzkonzept ist Verdoppelung und Vergleich.

  Integrität spielt jedoch auch eine wesentliche Rolle bei Transaktionssystemen in Bereichen wie beispielsweise dem Bankwesen, wo neben der Forderung nach konsistenter Datenhaltung auch eine hohe Verfügbarkeit gefordert ist. Daten dürfen weder verloren gehen, noch darf ihre Korrektheit durch Ausfälle gefährdet sein. Dagegen ist es i.a. tragbar, daß Daten kurzzeitig nicht verfügbar sind. Fehlertoleranzverfahren, die auf Zeitredundanz beruhen, z.B. die Wiederholung von Transaktionen, können hier also zum Einsatz kommen. Die Systeme können sogar um den Erdball verteilt sein (Reservierungssysteme). Sie stellen deshalb besonders hohe Anforderungen an ihre Verfügbarkeit. Eine für nicht fehlertolerante Rechner typische Verfügbarkeit von 99,7% reicht dann nicht aus. Dieser Wert mag zwar auf den ersten Blick hoch erscheinen, er bedeutet jedoch im Mittel eine Stunde Ausfall alle zwei Wochen bei einem 24-Stunden-Betrieb. Fehlertolerante Transaktionssysteme zeigen dagegen einen mittleren Abstand zwischen zwei Ausfällen im Bereich von Jahren.

Dabei beeinflußt das Anforderungsprofil die Wahl des Fehlertoleranzkonzepts (die Architektur des Systems) und damit auch den zu treibenden Aufwand.

Konzepte der Fehlertoleranzmaßnahmen

In diesem Abschnitt wollen wir die Möglichkeiten, Fehlertoleranz in Multiprozessorsystemen zu realisieren, näher betrachten. Dabei fällt auf, daß viele Fehlertoleranzmechanismen von der Antwort auf folgende drei wichtigen Systemdesign-Fragen bestimmt werden:

1) *Soll die Fehlertoleranz auf Gatter-Ebene, auf Chip-Ebene, auf Board-Ebene oder auf System-Ebene erfolgen?*

2) *Sollen die Einheiten lose oder fest gekoppelt sein?*

3) *Wie wird der Datenaustausch zwischen der Umwelt und dem Rechnersystem gestaltet?*

Wahl der Redundanzart und -Ebene

Anstelle der Hardware-Redundanz ist auch ein mehrmaliges Abarbeiten und anschließender Vergleich der Ergebnisse auf dem selben Rechner möglich ("Zeitredundanz", vgl. Abschnitt 1.4). Im Gegensatz dazu benutzen Systeme, die nach einem Fehler (*exception handling*, s. [CRIS3]) bei einem früheren, gespeicherten Zustand wieder aufsetzen ("Roll-back"), zwar auch Zeitredundanz, erlauben aber nicht die bei mehrmaligem Abarbeiten mögliche Kontrolle und evtl. Korrektur der Ergebnisse. Diese Art von Fehlertoleranz ist somit nur bei unkritischen Anwendungen verwendbar.

Betrachten wir nun Systeme mit überwiegender Hardware-Redundanz. Da aber die Fehlertoleranz auf Gatter-Ebene und auf Chip-Ebene Spezial-Hardware voraussetzt, die manchen wirtschaftlich motivierten Systemvorgaben (z.B. unserer Systemanforderung 3 aus Abschnitt 2.3.1)

widerspricht, empfiehlt sich in diesem Fall die Fehlertoleranz auf Board-Ebene. Die kleinste ersetzbare Einheit ist also ein Single-Board-Computer (SBC).

Diese Entscheidung bringt auch weitere Vorteile: Die Fehlertoleranz-Mechanismen sind leicht übertragbar (s. Anforderung 5 in Abschnitt 2.3.1), unabhängig von kleineren Modifikationen der Boards und beeinträchtigen nicht im Fehlerfall die Abarbeitungsgeschwindigkeit der Benutzerprogramme.

Kopplung der Prozessoren

Betrachten wir nun die Multiprozessor-Kopplungen aus Abschnitt 1.2 unter Fehlertoleranz-Gesichtspunkten. In Abbildung 3.0.1 und 3.0.2 sind zwei Multi-Prozessorsysteme mit fester und mit loser Kopplung gezeigt.

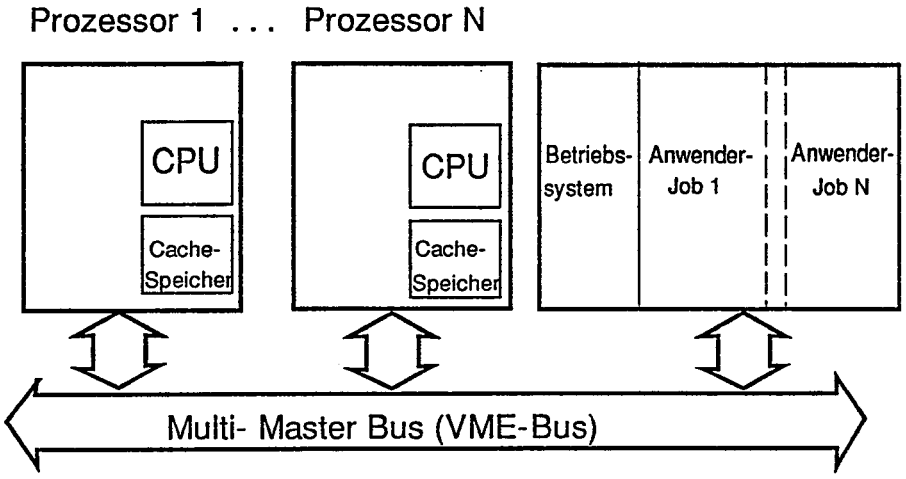


Abb. 3.0.1 Ein fest gekoppeltes Multi-Prozessor-System

Bei der festen Kopplung der Prozessoren in Abb. 3.0.1 arbeiten alle Prozessoren mit einem gemeinsamen Speicherbereich, in dem das Betriebssystem und die Benutzerprogramme liegen. Damit der Systembus nicht die Rechenleistung empfindlich begrenzt, ist zusätzlich auf jedem Prozessor-Board ein schneller Cache enthalten.

Diese Lösung hat folgende Nachteile:

- bei Ausfall des Systembusses fällt das Gesamtsystem aus
- defekte Prozessoren können (trotz eingebauter MMU) die globalen Daten sowie die privaten Daten der anderen Prozessoren korrumpieren und damit das System ebenfalls wertlos machen
- einzelne, defekte Speichereinheiten können, da alle Daten und Programme zentral gelagert sind, die Funktion aller Prozessoren behindern.
- Um Dateninkonsistenzen zwischen Cache und Hauptspeicher zu vermeiden, müssen entweder Standard-Programme (Lader, Debugger, etc) umgeschrieben werden, oder spezielle Hardwaremechanismen (s. Abschnitt 1.3.3) müssen zusätzlich bei dem Cache implementiert werden.

In dem lose gekoppelten System von Abb. 3.0.2 sind diese Nachteile vermieden.

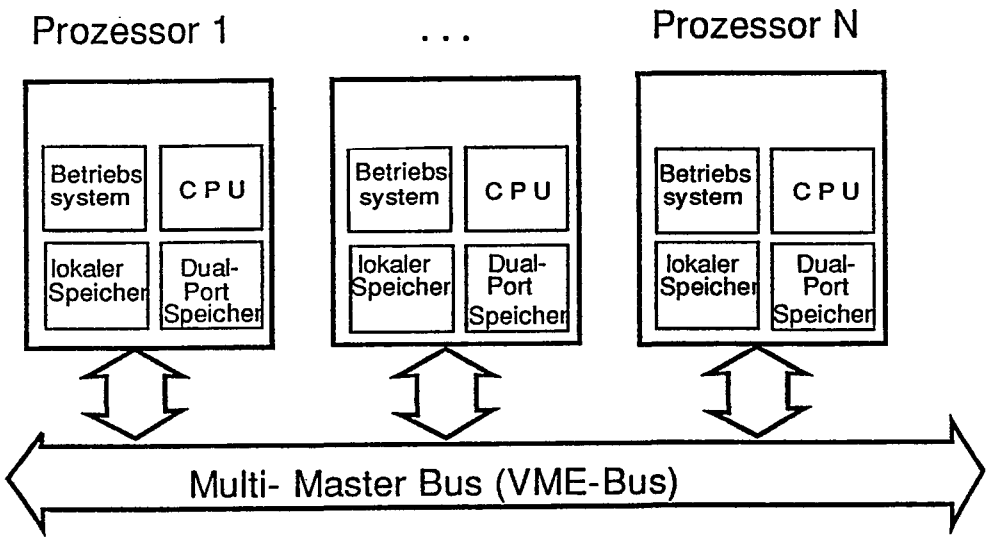


Abb. 3.0.2 Ein lose gekoppeltes Multi-Prozessor-System

Der Systembus wird nur noch zur Kommunikation (Austausch von Nachrichten zwischen den Prozessoren) verwendet; eventuell eingesetzte, zusätzliche Kommunikationswege lassen sich leicht für das Anwenderprogramm transparent einbinden.
Das Betriebssystem und das jeweilige Anwenderprogramm sind für jeden Prozessor direkt zugreifbar, ohne damit die anderen Prozessoren (Rechnerleistung!) oder seine Daten bei Defekten zu tangieren.
Der Nachteil dieser Lösung ist allerdings auch evident:

- Wurden vorher zur Koordination der Prozessoren bei globalen Ressourcen (z.B. I/O-Einheiten) Semaphoren oder Monitor-Konstrukte im gemeinsamen Speicher eingesetzt, so geschieht nun die Koordination durch Nachrichtenaustausch. Dies benötigt mehr Rechnerleistung.

Da sich aber Kommunikationsprotokolle sehr gut modular von den normalen Rechneraktivitäten abtrennen lassen, kann man den Nachteil relativ einfach durch einen besonderen Kommunikationsprozessor eliminieren.

Daten I/O
Bei der üblichen Lösung wird der Input (Analog- oder Digitaleingabe) von einem Interface empfangen und dann aktiv (DMA) oder passiv an die Prozessoren verteilt. Damit hängt aber wieder die Funktion des Gesamtsystems nur von einer einzigen Komponente ab.
Das Gleiche gilt auch für den Output.

Um dieses Problem zu vermeiden, lassen sich die Eingabedaten an jeden einzelnen SBC heranführen, indem man einen Broadcast-Mechanismus verwirklicht. Der denkbar einfachste Broadcast wird durch eine gemeinsame, physikalische Leitung erreicht. Beispiele dafür sind die Steuerdaten einer Prozeßregelung (s. Abb. 3.3.1) oder die Eingabedaten vom Benutzerterminal, dessen serielle Schnittstelle parallel mit allen seriellen Schnittstellen der SBC verbunden sein kann. Diese Lösung ist beispielsweise im ATTEMPTO-System (s. Abb. 2.3.1) verwirklicht.

3.1 Hardware - Konzepte

Die Hardware-Konzepte als traditioneller Ansatz zur Verwirklichung von Fehlertoleranz wurden meist in der Gruppe der hochzuverlässigen Systeme eingesetzt, um den Forderungen nach Realzeitverhalten und Fehlertoleranz gleichzeitig Rechnung zu tragen. Ein Beispiel dafür ist die Anwendung zur aktiven Steuerung dynamisch instabiler Flugzeuge (ehemaliges NASA Projekt), die extrem sicherheitskritisch ist und weniger als 10^{-9} Ausfälle/Stunde bei einem zehnstündigen Flug fordert. Hierfür wurden mehrere fehlertolerante Bordcomputer realisiert, deren Hardware-orientiertes Beispiel "FTMP" zuerst kurz besprochen werden soll. Danach folgt "Stratus", ein ebenso Hardware-Redundanz verwendender Ansatz zur Verwirklichung eines fehlertoleranten Transaktionssystems.
Beide Problemkreise können auch auf der Basis von Software fehlertolerant behandelt werden, wie die Projekte "SIFT" und "Tandem" im Abschnitt 3.2 danach zeigen werden.

FTMP

Das Multiprozessorsystem FTMP (*Fault Tolerant MultiProcessor*) wurde Ende der 70-ger Jahre von Draper Laboratories entwickelt [HOP], mit der Förderung des NASA Langley Research Center, dem Office of Naval Research und der US National Science Foundation.
Es benutzt Verdreifachung mit Mehrheitsentscheid, verbunden mit einer Rekonfigurationsmöglichkeit durch zusätzliche Reserveeinheiten (standby-Redundanz). Die ursprügliche, konzeptionelle Hardwarearchitektur ist in einer Übersicht in Abbildung 3.1.1 gezeigt.

Hardwarearchitektur

FTMP ist ein taktsynchron arbeitendes Multiprozessorsystem, bestehend aus Prozessor-, Speicher- und E/A-Modulen (Mil. Busstandard 1553), die über redundant ausgelegte, bitserielle Busse kommunizieren, siehe Abbildung 3.1.1.
Der realisierte Prototype weicht allerdings in einigen Punkten von dem Konzept ab. So wurden nicht für alle Einzelmodule und alle Überwachungseinheiten (*bus guardian*) extra Netzteile vorgesehen, sondern man beschränkte sich bei 10 Hauptmodulen auf vier unabhängige 24V Stromversorgungsleitungen. Die Hauptmodule bestehen aus einer Zusammenlegung eines Prozessors, Speichers, Uhr, E/A Modul und einem Netzteil (24V zu div. Spannungen) sowie zweier Busüberwachungseinheiten auf einer Platine.

Die zwei Busüberwachungseinheiten bei jedem Modul führen über die Daten von jeweils drei Bussen von drei Prozessoren oder Speichern einen Mehrheitsentscheid durch und vergleichen ihre beiden Entscheidungen miteinander. Im Differenzfall (Fehlerfall) werden Betriebssystemprozeduren angesprochen, die Tests zur Fehlerlokalisierung (Bus oder Modul) und eine anschließende Rekonfiguration auslösen, so daß (fast) immer eine Majorität von intakten Komponenten bei den Mehrheitsentscheidungen eingesetzt werden.

Softwarekonfiguration

Das Multiprozessorsystem arbeitet nach dem Schema eines speichergekoppelten Systems (s.Abschnitte 1.2, 1.3.3 und 2.1) und benutzt dazu spezielle Cache-Speicher bei den Prozessoren.
Die Betriebssystemsoftware ist für verschiedene Aufgaben verantwortlich. Dies sind beispielsweise

das Zuteilen und Aufsetzen von Tasks (Dispatching) aus der zentralen Warteschlange heraus, wobei die Bearbeitung der Programme von den Dispatcherprozeduren des Betriebssystemkerns

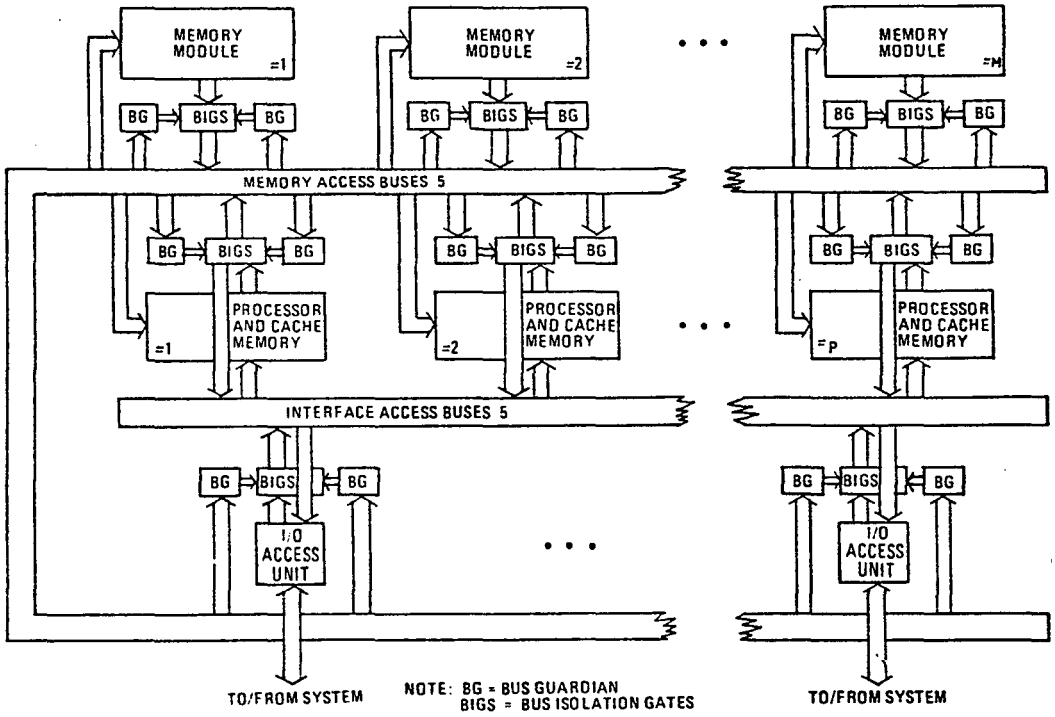


Abb. 3.1.1 konzeptionelle Hardwarestruktur von FTMP (aus [HOP])

nach dem Anziehungsprizip verwirklicht wird, wie dies etwa auch in PLURIBUS (Abschnitt 2.1) der Fall war. Ein anderes Beispiel ist die Betreuung einer globalen Ereignis-Warteschlange, die Abwicklung der I/O Operationen und das damit verbundene Error-logging, Fehlerdiagnose und Recovery sowie der Votierungsfunktionen für die Eingabe zum Anwendertask.
Desweiteren überprüft ein besonderer, für das Anwenderprogramm transparenter Mechanismus, ob die für den laufenden Task nötige Prozedur im Cache-Speicher ist. Wenn nicht, wird ein Teil des Cache-Speichers mit dem Code aus dem globalen Speicher überschrieben. Beim Rücksprung wird ebenso geprüft, ob der weiter nötige Code noch vorhanden ist oder nachgeladen werden muß. Damit wird der Cache-Speicher konsistent gehalten, ohne den Speicherbus zu sehr zu belasten.

Fehlertoleranzmechanismen
Die verschiedenen kritischen Bereiche des FTMP Systems wie das globale Zeitsystem, die Prozessorleistung und die globalen Speicherbereiche bauen auf dem Konzept einer hardwareunterstützten Majoritätsvotierung auf.
Da alle Operationen von der Datenerfassung und -glättung bis zum Majoritätsvergleich zeitsynchronisiert zwischen den beteiligten Einheiten ablaufen, ist die Einhaltung eines globalen Zeittaktes in FTMP sehr wichtig. Dies wird durch eine spezielle Hardwareschaltung erreicht, in der alle Taktsignale aller beteiligten Uhren phasengleich einrasten.
Zur Programmbearbeitung selbst finden sich jeweils Dreiergruppen (*Triaden*) von Prozessor/Cache-Speicher- Modulpaaren, Bussen und Globalspeichermodulen zusammen, wobei jedes Modul seine Ausgabedaten auf einen ihm zugeordneten Bus ausgibt, Eingabedaten ihm aber über die Bus-Triade dreifach zugeführt werden. Den beiden Buskoppelelementen (*bus*

guardian) der einzelnen Modulen kommt die Aufgabe des Mehrheitsentscheides zu; dabei können sie gleichzeitig defekte Triadenmodule lokalisieren und dies an eine Nachbartriade melden, die zuständig für die weitere Fehlerbehandlung ist: Ausgrenzen des Defektmoduls und Rekonfigurieren einer neuen funktionsfähigen Triade durch Zuschalten einer Reservekomponente.
Sind also 10 Module vorhanden, so wird mit 3 Triaden und einem Reserveelement die Leistung eines 3-Prozessorsystems verwirklicht.

Stratus

Ein typischer Vertreter integrer Rechensysteme und zugleich auch ein fehlertolerantes System, das eine nennenswerte kommerzielle Verbreitung im Datenbankbereich bei Transaktionssystemen gefunden hat, ist das Stratus ("*Continous Processing*") System, das auch von IBM als "System/88" vertrieben wird [HARR].

Hardwarearchitektur

Bei Stratus ist ein Duplexsystem durch HW-implementierte Fehlertoleranz realisiert: Jede Baugruppe einer Rechnereinheit (Abbildung 3.1.2) ist gedoppelt, ebenso der sie verbindende Bus und die Platteneinheiten. Die Baugruppen sind selbstprüfend und zeigen ein "Fail Stop"-Verhalten, d.h. sie isolieren sich selbst vom Bus nach der Entdeckung eines eigenen Fehlers. Dazu enthalten sie jeweils zwei Prozessoren, die sich taktsynchron überwachen.

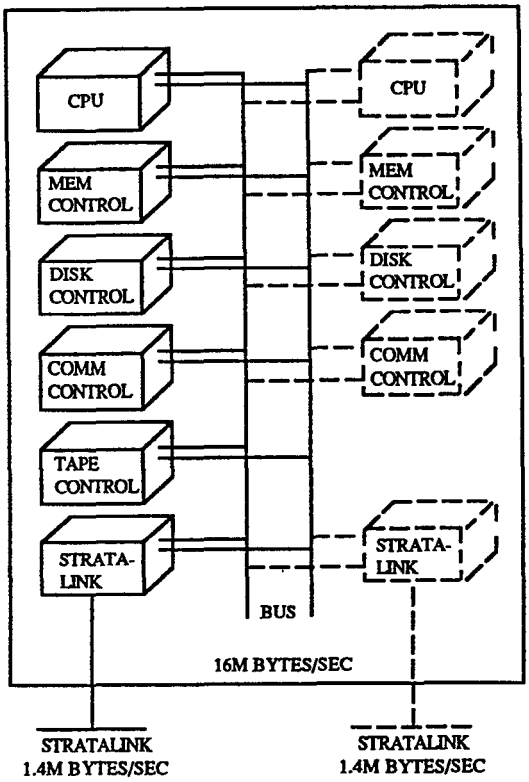


Abb. 3.1.2 Das Stratus - System

Die erwähnte Fail Stop-Eigenschaft wird bei Stratus durch Verdoppelung und Vergleich aller HW-Funktionseinheiten auf der Baugruppe selbst erreicht. Der Aufwand für diese Form der

HW-Redundanz, kurz "pair and spare" genannt, beträgt mehr als das Vierfache eines nicht fehlertoleranten Systems, wobei die Redundanznutzung eher statischen Charakter besitzt, wie auch das Systemverhalten dem maskierender Systeme ähnelt. Da die Prozessoren auf jeder Baugruppe nochmals in Betriebssystemfunktionsprozessor und Anwenderprozessor aufgeteilt sind, sind insgesamt für einen Prozessor derer acht im System vorhanden, allerdings bei (nominell) doppelter Abarbeitungsgeschwindigkeit.

Fehlertoleranzmaßnahmen
Im Fehlerfall führt die verbleibende Hälfte des Paares (die zweite Baugruppe) den Betrieb unterbrechungsfrei fort, während die defekte Baugruppe durch ein Diagnoseprogramm getestet wird. Wird dabei ein transienter Fehler diagnostiziert, kann die Baugruppe wieder durch Resynchronisation mit Hilfe des um spezielle Betriebssystemprozeduren erweiterten UNIX in den laufenden Betrieb reintegriert werden. Dazu wird der Status der intakten Baugruppe (Prozessorregisterinhalte, etc) ausgelesen und auf beide Baugruppen zurückgeschrieben.
Liegt kein transienter Fehler vor, so wird auf dem Modul eine Signallampe angestellt und es muß eine manuelle Reparatur (Platinenaustausch) erfolgen, die "on line", d.h ohne Unterbrechung des nun nicht mehr fehlertoleranten Rechenbetriebs möglich ist.

Ein Ferndiagnosesystem erlaubt außerdem, die Fehlermeldungen und Ausfallsdiagnosen über Modem und Telefonleitung zu einem Hersteller-Servicezentrum zu leiten und eine automatische Lieferung der nötigen Ersatzteile zu veranlassen. Damit können die Zeiträume, bei denen ein Ausfall einen Systemstillstand bewirken könnte, relativ kurz gehalten werden.

3.2 Software - Konzepte

Einer der interessantesten Möglichkeiten, Fehlertoleranz in ein Computersystem einzuführen, besteht darin, keine spezielle, fehlertolerante Hardware zu entwickeln, sondern die Fehlertoleranz durch eine geschickte Kombination aus Hardware-Redundanz und Software zu verwirklichen. Beispiele dafür sind fehlertolerante Systeme mit vielen Prozessoren wie sie für die künstliche Intelligenz (s. [MAE3] und Kapitel 4) eingesetzt werden.
In dem folgenden Abschnitt sind zwei mittlerweile "klassische" Systeme beschrieben, die mit unterschiedlichen Fehlertoleranzmethoden für zwei unterschiedliche Anwendungsgebiete fehlertoleranten Rechnerbetrieb überwiegend mit Hilfe von geeigneter Software implementieren.

SIFT
Eines der ersten Projekte, die unter Verwendung von Standardkomponenten nur mit Hilfe von Software ein Rechnersystem hochzuverlässig machen sollte, ist das SIFT-Projekt (*Software Implemented Fault Tolerance*) des Stanford Research Institutes [WEN1].
Dieser fehlertolerante Bordcomputer auf Softwarebasis war sogar für die Anwendung bei der aktiven Steuerung dynamisch instabiler Flugzeuge (wie im vorigen Abschnitt besprochen) konzipiert. Die Fehlertoleranz-Spezifikation war dabei wie bei FTMP ziemlich hoch: Während eines Flugs von 10 Stunden mußte eine Ausfallwahrscheinlichkeit von weniger als 10^{-9} pro Stunde garantiert werden.

Hardwarearchitektur

In der ersten Version waren je 5 Rechenmodule, I/O Module und Busse nach dem "Vorzimmer"-Modell aus Abschnitt 1.2 vorgesehen [WEN2]. Inzwischen wurde SIFT aus gleichartigen, autonomen Computer-Modulen, die durch elektrisch isolierte Busse (Defekt-Isolierung!) vollständig miteinander vernetzt sind, realisiert [MELL], s. Abbildung 3.2.1. Damit ist ein Broadcast unter allen Prozessoren möglich.

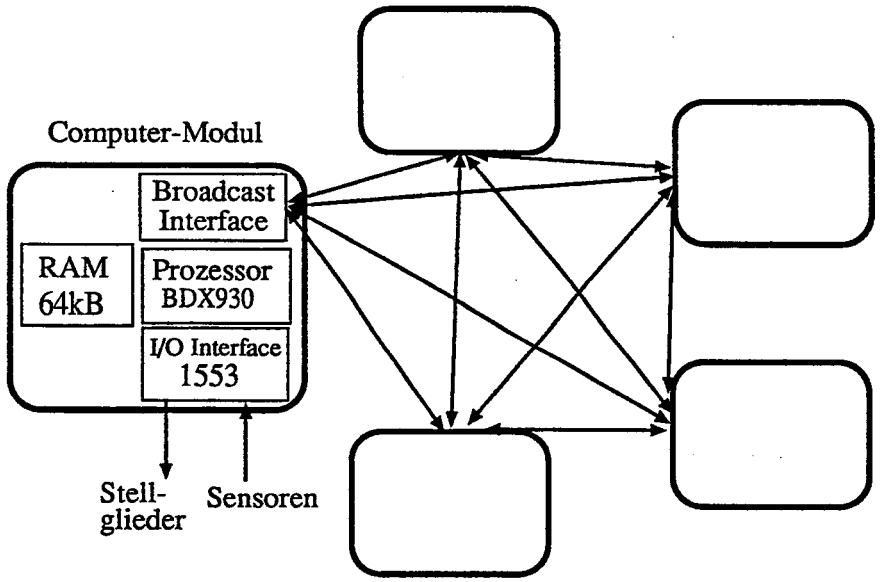


Abb. 3.2.1 Überblick über die SIFT-Hardwarekonfiguration

Die Busse waren, um eine Fehlerpropagierung einzuschränken, als reine READ-ONLY Busse konzipiert, deren Kontrollen durch time-out abgesichert sein sollten. In der realisierten Version wurden jedoch reine WRITE-ONLY Broadcast-Verbindungen eingesetzt, die aktiv schreibende Prozesse erfordern.

Softwarekonfiguration

Die Kopien der Anwendungsprogramme (*tasks*) sowie Kopien eines globalen Fehlerbehandlungs- und Rekonfigurationsprogramms (*global executive task*) und seine lokalen Zusatzprogramme (*error reporting task, local reconfiguration task*) werden auf verschiedenen Prozessor-Speicher-Modulen ausgeführt und kommunizieren miteinander. Dazu existiert auf jedem Prozessor-Speicher-Modul ein lokales Betriebssystem (*local executive*), das Systemtabellen-geführte Kommunikationsdienste anbietet. Die Systemtabellen werden bei einer Rekonfiguration vom local reconfiguration task nach Maßgabe des global executive task geändert. Die Anwendungsprogramme müssen (und dürfen!) also nicht wissen, auf welchem Prozessor-Speicher Modul ihr Kommunikationspartner sich befindet.

Die Konsistenz der Eingabedaten (Sensorwerte) wird mit Hilfe eines speziellen, interaktive Konsistenz genannten Mechanismus [PEA] erreicht, so daß von allen replizierten Tasks auch die gleichen Ergebnisse berechnet werden.

Da die hochzuverlässigen Systeme meist (und hier besonders!) mit harten Real-time

Forderungen verknüpft sind, wurde ein besonderer, prioritätsorientierter Schedule entworfen, um die verschiedenen Tasks zu synchronisieren. Dabei dienen verschiedene, ganzzahlige Multiple einer Grundperiode (ca. 20ms) als Synchronisationsrahmen der Tasks (mind. 2ms je Taskabschnitt). Je kleiner der multiple Faktor, desto höher die Priorität des auszuführenden Tasks.
Im Unterschied zu normalen, prioritätsgesteuerten Schedulen mit einer maximalen, sicheren Auslastung von ca. 70% erreicht obige Strategie [WEN2] die maximale Auslastung der deadline-orientierten Scheduler von 100%, die bei dieser Anwendung durch ihren großen Aufwand nicht in Betracht kommen.

Fehlertoleranzmaßnahmen
Als Hauptmaßnahme zur Fehlertoleranz wurde hier ebenfalls, wie in FTMP, eine Verdreifachung mit Mehrheitsentscheid, verbunden mit einer Rekonfigurationsmöglichkeit durch zusätzliche Reserveeinheiten (standby- Redundanz), benutzt. Ein Vergleich FTMP mit SIFT ist dabei besonders gut geeignet, um die unterschiedliche Implementierung von Fehlertoleranzkonzepten bei gleichem Anforderungsprofil zu beschreiben.

Das Bussystem dient zum Austausch von Broadcast-Nachrichten, die in der Regel Daten von Rechnereinheiten enthalten, die an der gleichen Aufgabe arbeiten. Über deren Ergebnisse (Soll-Zustand der Flugzeugsteuerung) ist nach jeder Iteration mit neuen Sensordaten eine Mehrheitsentscheidung zu fällen. Die eigentlichen Kommunikationspartner sind "Tasks", strukturelle Einheiten des Anwenderprogramms, die durch das Betriebssystem auf verschiedenen Rechnereinheiten repliziert werden und die nahezu zeitgleich operieren. Die geforderte Taskstruktur der Anwender-Software definiert Synchronisationszeitpunkte (s.o), die wegen der Asynchronität der Rechner um bis zu 50 µs differieren können. Für eine korrekte Zeitsynchronisierung müssen, so läßt es sich zeigen, beim Auftreten von beliebigen, "bösartigen" Fehlern in einer Uhr mindestens 3 weitere Uhren korrekt funktionieren [LAM1]. Die Aufgabe des Synchronisierens und Votierens übernimmt das jeweilige lokale Betriebssystem, das spezielle Prozeduren anbietet, bei denen nicht nur das einfache Lesen einer Nachricht (Puffer) möglich ist, sondern auch diese Nachricht vorher als Ergebnis eines Votierens über multiple Nachrichten verschiedener Tasks ermittelt wird. Durch Verdreifachung der Tasks ist so TMR-Betrieb möglich, darüberhinaus auch NMR, da der Redundanzgrad wählbar ist.

Gegenüber FTMP ist die variable Ausnutzung bzw. Definition der Redundanz ein wesentlicher Vorteil der Software (SW)- implementierten Fehlertoleranz, insbesondere bei degradierenden Systemen; die Hardware (HW)- Implementation ist dagegen besser geeignet für harte Echtzeitanforderungen. Der Maskierer ist eine zentral die Zuverlässigkeit bestimmende Komponente. Für Hardwarevotierer bzw. -Umschaltnetzwerke existieren bekannte Lösungen; bei Software-Implementationen wird in der Regel eine (formale) Verifikation der verwendeten Fehlertoleranz-SW gefordert. Desweiteren sind hier - vor allem bei dynamischer Redundanz - zusätzliche Aufwendungen nötig, um eine weitgehende Transparenz der Fehlertoleranz-Maßnahmen für die Anwender-Software zu ermöglichen.

Verifikation des Systems
Das SIFT Projekt war eines der ersten Projekte, bei dem versucht wurde, das korrekte Funktionieren der Anlage bei auftretenden Fehlern formal zu beweisen. Bedenkt man die

geforderte, geringe Ausfallwahrscheinlichkeit, so ist angesichts der Alternative, 114 155 Jahre auf einen Ausfall warten zu müssen, die Möglichkeiten für ein statistisches Experiment nicht gegeben. Die Verifikation des SIFT-Systems beruht auf einer Modellhierarchie, die in Abbildung 3.2.2 wiedergegeben ist.

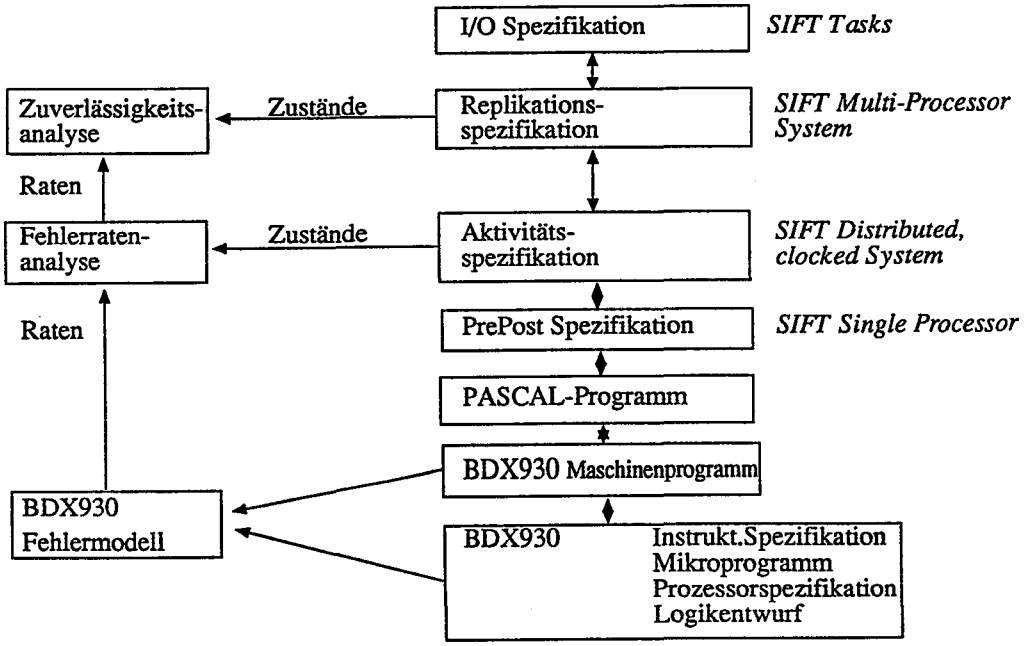


Abb. 3.2.2 Hierarchie der Modelle für SIFT

Jede Modellebene beschreibt dabei vergröbert die Funktionen der nächstunteren Ebene. Ist es möglich, den Korrektheitsbeweis der Zustandsübergänge auf der obersten Ebene zu führen sowie stufenweise jeweils in der unteren Schicht die Korrektheit nur innerhalb derjenigen Zustandsübergänge zu zeigen, die auf einen einzigen, verallgemeinerten Zustand der darüberliegenden Schicht abgebildet werden, so ist damit der Korrektheitsbeweis für das gesamte System geführt.
M. Melliar-Smith und R. Schwartz zeigten in ihrer Arbeit [MELL], wie eine Verifikation der Vorgabe für die Ausfallraten der verschiedenen Abstraktionsstufen der Modelle in Abbildung 3.2.2 durchzuführen ist. Der Nachweis bedient sich dabei zwei verschiedener Wege.

Zum einen muß man, wie dies schematisch in Abbildung 3.2.2 rechts gezeigt ist, die Korrektheit der I/O-Systemfunktion beweisen, falls das System in einem funktionstüchtigen Zustand (*safe state*) ist, d.h. eine für die Fehlertoleranzmechanismen ausreichende Anzahl von Prozessoren, Speicher, I/O Module und Buscontroller korrekt funktioniert. Der funktionstüchtige Zustand wird durch den Wert WAHR eines logischen Prädikats `system_safe` ausgedrückt.

Zum anderen muß man (Abb.3.2.2 links) ausrechnen, mit welcher Wahrscheinlichkeit ein solcher sicherer Zustand *nicht* auftritt. Dafür werden alle Hardwarefehler des Fehlermodells in zwei Klassen eingeteilt: diejenigen, deren Existenz inkonsistente Systemdaten bei den anderen Prozessoren hervorruft (*byzantinische Fehler*), und diejenigen, die dies nicht tun.
Dieses Fehlermodell, verbunden mit den tatsächlichen Auftrittswahrscheinlichkeiten der verschiedenen Hardwarefehler der Einzelkomponenten, führt zu Ausfallraten durch verschiedene Fehlerklassen. Beachtet man noch, welche Auswirkungen (Fehler) diese Ausfälle nach sich ziehen,

so ergeben sich die Fehlerraten der möglichen Fehler des Fehlermodells bzw. die einzelnen Auftrittswahrscheinlichkeiten der verschiedenen Systemzustände *(h,d,f)*. Dabei werden die Übergänge mit ihren Übergangswahrscheinlichkeiten zwischen den *f* aufgetretenen, *d* bemerkten und *h* bereits behandelten Ausfällen von Prozessormodulen durch einen Semi-Markov Prozess beschrieben [WEN2].

Betrachten wir nun die formale Verifikation des SIFT-Systems genauer, um an diesem Beispiel die Systematik eines für große Systeme notwendigen, gegliederten, logischen Korrektheitsbeweises besser zu verstehen. Dazu beschränken wir uns auf wenige, ausgewählte Beispiele einer jeden Schicht, die in einer PROLOG-ähnlichen Syntax präsentiert werden, obwohl damals eine spezielle, parallel zum SIFT-Design entwickelte Spezifikationssprache STP verwendet wurde [MELL].

SIFT I/O Spezifikation

Für jeden Anwenderprozeß gibt es eine anwenderspezifizierte `function()`, deren Programmierung als korrekt vorausgesetzt wird (anwenderbedingte Verifikation). Ist diese Funktion gegeben, so definiert das zentrale I/O Axiom auf oberster Ebene die Korrektheit der Ausführung. Sei die i-te Iteration der Task a mit dem Prädikat `it_of(a,i)` notiert, das `TRUE` während der Ausführung ist, und mit `it_to(b,it_of(a,i))` die Ausführung der Input Task b, deren Daten Input für die i-te Iteration der Task a werden.

I/O AXIOM	Kommentar
`on_during(a,i)`	| Result wird tatsächlich errechnet
AND `task_save_during(a,i)`	| Task a ist *save* bei der i-ten Iteration
AND ∀ b element_of *a* `Inputs(a)`	| Alle Tasks b, die Input für a bereitstellen
`|Result(b, it_to(b,it_of(a,i)))| = 1`	| Diejenige Iteration von b, deren Ergebnis Input
	| für die *i*-te Iteration von *a* ist, hat als Resultat
	| nur ein einziges Ergebnis,
⇒ `Result(a,i)=[apply (function(a), v)]`	| wobei das Resultat durch die Anwendung
	| einer Funktion auf Daten definiert wird.
`v = <v,t>,`	| Die Daten bilden eine Tabelle, in der jeder
`t element_of Inputs(a)`	| Inputtask *t* dasjenige Ergebnis zugeordnet
AND `v element_of Result(t,it_to(t,it_of(a,i)))`	| wurde, das von ihr für die *i*-te
	| Iteration von *a* produziert wurde.

Für die Task, die die verschiedenen Sensordaten für andere Tasks in einem interaktiven Prozeß konsistent aufbereitet, ist die `function()` die Identität. Allerdings kommt hier noch die Bedingung

`interact_consist (a) AND save_interact_consist (a,i) ⇒ |Result(a,i)| = 1`

hinzu: eine interaktive Konsistenztask, die *save* ist, produziert ebenfalls nur *ein* Ergebnis.
Bei der Ausführung der Tasks wird angenommen, daß sie eine bestimmte Zeit (Ausführungszeitfenster) benötigen: `EWindow(a,i) := [beg(it_of(a,i)), end(it_of(a,i)]`

SIFT Replikationsspezifikation

Bei der Spezifikation der höheren I/O Funktionen werden Primitive benutzt, die auf Begriffen wie "Multiprozessoren" und weiteren Zeitfenstern beruhen, die für die Fehlertoleranz-Mehrfachausführung der Task nötig sind. Die Existenz dieser Details wurde in eine der I/O Schicht zugrunde

liegende Schicht der Replikationsspezifikation verborgen.
Mit den Definitionen

`poll_for(p,a,i)`	TRUE, wenn p einer der Prozessoren ist, der die i-te Iteration von a ausführt
`poll_for (a,i)`	Die Menge aller derartigen Prozessoren

wird ein Prädikat der I/O Schicht spezifiziert

`on_during(a,i)` := ∃ p mit `poll_for(p,a,i)` = TRUE

Mit den weiteren Definitionen

$S(p,T_{12})$	TRUE, wenn der Prozessor p im Intervall $[t_1, t_2]$ *save* ist.
$S(T_{12})$	Die Menge aller derartigen Prozessoren

Ein Datenfenster reicht vom Erstellen der Inputdaten durch Task b bis zur Verarbeitung durch Task a:

```
DWindow(b,it_of(a,i)) :=   [beg(it_to(it_of(a,i))), end(it_of(a,i)]
```

wird nun das Prädikat `task_save_during(a,i)` der I/O Schicht durch die *safe*-Eigenschaft der an der Majoritätsentscheidung beteiligten Tasks spezifiziert:

```
task_save_during(a,i) :=
        2 |poll_for(a,i) ∩ S(DWindow(b,it_of(a,i)))|  >  |poll_for(a,i)|
OR      NOT on_during(a,i)
```

Entweder gibt es eine Majorität der *save* Tasks oder die Task wird nicht bearbeitet.
Auch die Majoritätsentscheidung selbst wird mit den definierten Vorbedingungen spezifiziert:

```
     S(p, EWindow(b,j))                          | Für den save Prozessor p werden
AND  {1≤y≤Result_Size(b)}                        | alle Ergebniskomponenten mit Index y errechnet, indem
⇒(Result_in(p,b,j))[y] = majority ( v̄[y])  | die Mehrheit genommen wird
mit  v̄ = {(v,q) |                               | von den Ergebniswerten,
          poll_for(q,j,b)                        | die Prozessor q in der j-ten Iteration mit b errechnet hat
     AND (v=Result_on_in(q,p,b,j))}              | UND vom Prozessormodul p empfangen wurden.
```

Damit läßt sich nun in der Replikationsschicht die Funktion `Result(a,i)` lokal auf einem Prozessor für die höhere I/O Schicht spezifizieren als Menge aller Ergebniswerte, die von intakten Prozessoren votiert erlangt wurden:

```
Result(a,i) := {v | ∃p mit S(p,EWindow(a,i)) AND (v=Result_in(a,i))}
```

Für den Beweis der I/O Eigenschaften ("Theoreme") muß nun gezeigt werden, daß mit den Replikationsaxiomen daraus die I/O Spezifikationen folgen bzw. konsistent dazu sind. Der mechanische Beweis umfaßte bei SIFT ca. 22 Unterbeweise und insgesamt 106 Instantionen [MELL].

SIFT Aktivitätsspezifikation

Auf dieser Ebene wurden sehr konkrete Spezifikationen erstellt über die unabhängige, lokale Prozessoraktionen (Betriebssystemfunktionen) bezüglich der Zeit, der Systemkonfiguration und des Taskscheduling. Dazu wurde das Ausführungsfenster (*execution window*) in einen Ausführungsteil und einen Votierungsteil unterteilt.
Die lokale Implementierung (Spezifikation) der Primitive `Result_on_in(.)`, die zur Spezifizierung von `Result_in(.)` und damit für `Result(.)` in der Replikationsschicht verwendet wurde, lautet hier

```
v=Result_on_in(p,q,a,i)                  | Das votierte Resultat in q, das auf p für die i-te Iteration
```

```
                                          | von Task a erstellt wurde:
:= ∀ y,t                                  | Für alle Resultatkomponenten y und alle Taskscheiben t
     beg(it_of(a,i)) ≤ t < end(it_of(a,i)) | die in der i-ten Iteration von Task a liegen
     AND 1≤y≤Result_Size(a)               | UND Resultatkomponenten der Task a sind
     AND <vote,k,y>                       | UND deren Votierungstask
     element_of sched(config(t,q),t,q)    | im Taskabschnitt scheduled wurde
⇒v[y] = (datafile_in_for_on_at( q,a,p,start(t,s))) [y]
                                          | sind die Daten aus dem lokalen Puffer von q für Task a zu
                                          | nehmen, der von p vorher aufgefüllt worden ist.
```

Entsprechend werden auch die anderen Primitive wie S(.), `poll_for_of(.)` etc sowie die Zeitbedingungen der Uhren (Synchronisation) spezifiziert.

SIFT PrePost und Imperative Schichten

Auf den bisherigen Ebenen wurde beschrieben, wie sich die Ergebnisse durch periodisches Votieren replizierter Tasks innerhalb eines diskreten Zeitrahmens ergeben.
Es bleibt nun noch übrig, den Beweisweg für die in PASCAL geschriebene Software zu skizzieren, die die genannten Funktionen tatsächlich implementiert.
Dazu wird auf der PrePost-Ebene zunächst jede in PASCAL benutzte Datenstruktur mit einem möglichen Prozessorstatus <p,t> (ProzessorId und SubFrameValue) indiziert. Alle Aktivitäten der sequentiellen PASCAL-Programme werden modellartig durch die Zustandsüberführungen der Daten und des Prozessorstatus von einer Vorbedingung (*Precondition*) zu einer Nachbedingung (*Postcondition*) charakterisiert, beispielsweise für die vorher erwähnte `<vote,k,y>` Aktivität. Dieser Übergang Precondition→Postcondition durch eine Funktion ist einerseits leicht im Beschreibungsformalismus der höheren Spezifikationsebenen zu fassen und ermöglicht andererseits einen einfachen Übergang zur konventionellen Verifikationstechnik sequentieller, imperativer Programme.

Eine besondere Rolle nimmt dabei der Scheduler des lokalen Betriebssystemkerns ein. Lassen sich die sequentiellen, imperativen Programmteile noch dadurch verifizieren, daß der Prozessorstatus <p,t> und die korrekte Prozessorfunktion implizit verborgen bleiben und eine korrekte PASCAL-Maschine (d.h. korrekter Compiler und Maschinencode) vorausgesetzt wird, so ist dies beim Scheduler nicht mehr so einfach möglich. Hier wird die Funktion `next(<p,t>) =: <p,t+1>` der PrePost-Ebene auf die imperative PASCAL-Prozedur `dispatcher()` abgebildet, die bei jedem Zeittick per Interrrupt aufgerufen wird. Für den Nachweis der Zulässigkeit dieser Abbildung mußten zwei Eigenschaften bewiesen werden:

- Die Prozedur wird durch keinen Interrupt selbst wieder unterbrochen.
- Es tritt keine Interferenz mit anderen Prozessoren beim Zugriff auf interne und externe Daten auf (*critical regions*, s. Abschnitt 1.3.2).

Die erste Eigenschaft wurde durch eine Zeitanalyse des Maschinencodes und die zweite durch einen Beweis auf der Aktivitäts-Spezifikationsebene gesichert.

Damit ist (im Unterschied zum FTMP-System) die Verifikation der SIFT-Mechanismen für Fehlertoleranz auf verschiedenen Ebenen durchgeführt worden. Allerdings bedeutet dies nicht unbedingt vollständige, garantierte Fehlerfreiheit. Bereits in der Spezifikation können sich, wie dies in Abschnitt 3.4.5 näher ausgeführt werden wird, typische Entwurfsfehler verbergen; ebenso wie in der vom Anwender programmierten `function()`, die bedarfsgemäß den wechselnden Anwendungen angepaßt werden muß.

Tandem

Als zweiten typischen Vertreter für integre Rechensysteme ebenfalls mit einer nennenswerten kommerziellen Verbreitung sei das Tandem *"Non Stop"* - System erwähnt. Im Unterschied zu dem SIFT System liegt der Hauptaspekt dieses speziellen, für Transaktionen zweckbestimmten Systems nicht auf einer geringen Ausfallwahrscheinlichkeit bei harten Real-Time Anforderungen, sondern auf einer erhöhten Datenintegrität, Datenkonsistenz und Verfügbarkeit auch bei auftretenden Defekten.

Die Fehlertoleranzmechanismen sind hauptsächlich in Software ausgeführt und bestehen nach Ausfall eines Prozessors aus einem Wiederaufsetzen der beteiligten Prozesse auf einen früheren, gesicherten Zustand (*roll-back*, s. Abschnitt 1.4.1), wobei die im System vorhandene Hardwareredundanz nur für eine Rekonfiguration und nicht für eine Ergebniskontrolle (Maskierung) genutzt wird. Voraussetzung für eine korrekte Funktion des Systems ist also die Existenz von *fail-save* Hardware.

Hardwarearchitektur

Das Tandem-System zeigt sich mit seiner Software-implementierten Fehlertoleranz im Fehlerfall wesentlich flexibler als die Hardware-orientierten Systeme. Die "klassischen" Systeme (Tandem16) bestehen aus maximal 16 Prozessoren (16Bit) mit je maximal 512kB RAM, die über zwei Interprozessorbusse (16Bit Datenbreite, ca. 10MB/s) miteinander gekoppelt sind. Jeweils zwei Prozessoren haben einen Zugang zu I/O Controllern über ihren I/O Kanal (4MB/s), so daß jeder I/O Controller bei Ausfall eines Prozessors von jedem anderen Prozessor entweder direkt über den I/O Kanal oder indirekt über einen Interprozessorbus und den zweiten, direkt verbundenen Prozessor und dessen I/O Kanal Befehle erhalten kann. Damit können beispielsweise die Platteneinheiten ebenfalls zweifach redundant vorhanden sein und als Spiegelplatten betrieben werden.
In Abbildung 3.2.3 ist ein Überblick aus [KATZ] über die Hardware des Systems gegeben.
Jeder Prozessor hat eine eigene Spannungsversorgung. Da logisch zu jedem Bus auch eine Spannungsversorgung gehört, ist die Funktion jedes I/O Controllers über eine Diodenkopplung der

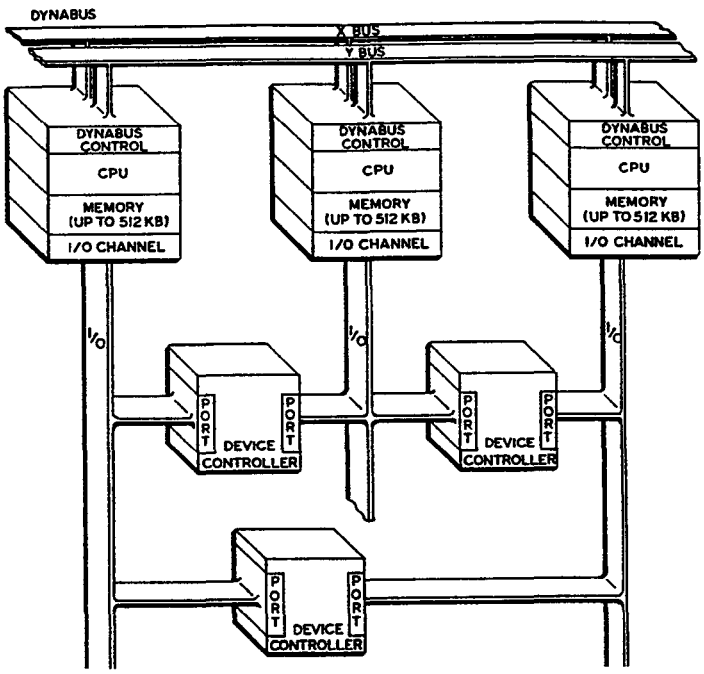


Abb. 3.2.3 Übersicht über die Tandem Hardware

beiden Spannungsversorgungseingänge gesichert.

Softwarekonfiguration
Die Systemsoftware ist in lokaler Replikation vorhanden. Bei jedem Prozessor gibt es Systemtafeln, die den Zugang zu den Interprozessorbussen (BusId, Adressen der Datenpuffer etc) und den I/O Kanälen (Pufferadressen etc) regeln und so eine Rekonfiguration ermöglichen.
Mit Hilfe des Betriebssystems kann für jeden Anwenderprozeß ein Ersatzprozeß auf einer anderen Rechnereinheit erzeugt werden; auch Betriebssystemprozesse können gedoppelt sein. Um diese Ersatzprozesse jederzeit aktivieren zu können, müssen sie zusätzlich über die laufenden Aktivitäten des Primärprozesses informiert werden. Die Häufigkeit solcher Nachrichten bestimmt die Praktikabilität eines solchen Konzepts; deshalb wurden im Tandem-System pro Transaktion nur drei Rücksetzpunkte gesetzt und deren Daten zum Ersatzprozeß gesendet. Bei der Erstellung von Anwenderprogrammen ist eine explizite Berücksichtigung der Fehlertoleranz durch Programmierung dieser Rücksetzpunkte erforderlich. Allerdings wird auf einer höheren Ebene weitgehende Transparenz für den Benutzer durch die vorhandene Datenbank-Software erzielt.

Fehlertoleranzmaßnahmen
Bei Tandem findet eine Fehlerdiagnose auf der globalen Systemebene statt: alle Rechnereinheiten senden zeitperiodisch Lebenszeichen ("*I am alive*"-Botschaften) an alle anderen. Beim Ausbleiben, beispielsweise nach einem nicht erfolgreichen Selbsttest, wird der Defekt dieser Systemkomponenten angenommen.

Zur Fehlerbehebung wird der Ersatzprozeß in einen zuvor durch den Primärprozeß definierten Zustand (*Rücksetzpunkt*) versetzt (*roll-back:* s. Abschnitt 1.4) und übernimmt dann dessen Funktion. Außerdem generiert er, nachdem der ausgefallene Rechner wieder verfügbar ist, dort für sich selbst einen neuen Ersatzprozeß. Ist der defekte Prozessor repariert worden, so wird automatisch der Ersatzprozeß auf den alten Rechner verlagert und Primärprozeß und Ersatzprozeß tauschen die Rollen, so daß die ursprüngliche Konfiguration wieder hergestellt ist.

3.3 Real-Time Aspekte

Vielfach sollen auch Systeme, die Daten in Real-Time zur Steuerung von industriellen Prozessen oder anderen Regeleinrichtungen (Flugzeugsteuerungen, s. FTMP und SIFT in Abschnitt 3.1 und 3.2) fehlertolerant arbeiten. Hierbei sind allerdings besondere Probleme zu beachten, die im folgenden Abschnitt näher betrachtet werden sollen. Ein Überblick über fehlertolerante, industriell genutzte Rechensysteme und ihre Probleme ist in [KIRR] zu finden.

Die zwei Hauptprobleme dieser speziellen Gruppe von Rechnern bestehen darin, einerseits Sensordaten möglichst fehlertolerant zu erfassen, auszuwerten und Stellglieder damit anzusteuern und andererseits dies auch in einer bestimmten, maximalen Zeitspanne durchzuführen. Beginnen wir mit der Datenerfassung und -verteilung.

3.3.1 Konsistenz der Sensordaten

Die einfachste Möglichkeit, Daten von Sensoren an verschiedene, multiple Recheneinheiten zu verteilen, besteht im Einsatz eines einheitlichen Datenbusses, wie er im vorigen Abschnitt vorgeschlagen wurde. Für besonders fehlertolerante Prozeß-Systeme ist ein derartiger singulärer Datenbus aber sicher bedenklich. Abbildung 3.3.1 zeigt eine Alternative dazu, bei der Daten vom gleichen Prozeß, aber von verschiedenen Sensoren ins System gehen und mit getrennten Leitungen wieder auf verschiedene Effektoren wirken. Überlagern sich diese Effektoren (Stellglieder) nur in ihrer Wirkung und sind nicht voneinander abhängig, so kann für die Gesamtregelung unter Umständen auch der (fail-save) Ausfall eines Effektors toleriert werden.

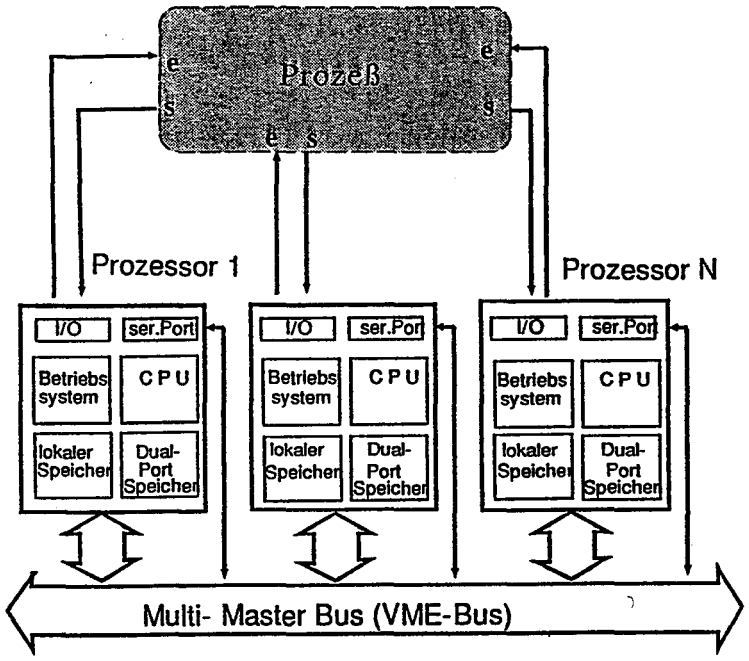


Abb. 3.3.1 Ein fehlertolerantes Prozeß-I/O System

Allerdings impliziert dieses Konzept das Problem, bei Vorlage unterschiedlicher Sensordaten

(verschiedene Analog/Digital-Wandler Daten) zu untersuchen, ob die Abweichungen unwesentlich sind (Offset-Drifts etc), und in diesem Fall die Daten zu vereinheitlichen, oder ob es sich um Fehler der I/O Einheiten handelt. Dieses Problem ist im SIFT System durch den Interaktiven Konsistenz-Task gelöst worden. Im fehlertoleranten FUTURE-System ist obiges Problem näher untersucht worden und soll an dieser Stelle deshalb kurz dargestellt werden.

FUTURE

Hauptziel des FUTURE-Systems (**F**ault-tolerant **U**ser friendly system with **T**askspecific **U**tilization of **RE**dundancy) ist, für den Benutzer nur soviel Aufwand an Fehlertoleranz zu treiben, wie er tatsächlich benötigt. Dazu werden verschiedene Klassen für die Ausführung der Anwendertasks definiert, die unterschiedlich große Anforderungen an Hardwareredundanz stellen [FÄR]. Es wird ein echtzeitfähiger Betriebssystemkern verwendet, ohne daß dies aber eine stark synchrone, globale Zeitbasis wie im danach vorgestellten MARS-System impliziert.

Hardwarearchitektur

Das Rechnerkernsystem von FUTURE besteht aus insgesamt vier Mikrorechnern (PDP11/03), die mit Broadcastsystemen für I/O und Interprozessorkommunikation ausgestattet sind [DEMM]. In Abbildung 3.3.1 ist eine Übersicht gezeigt.

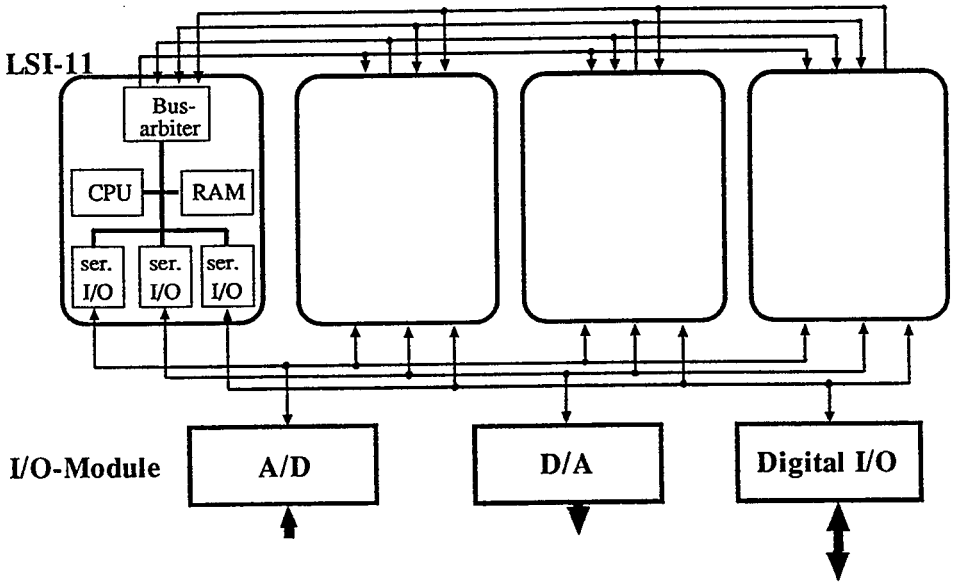


Abb. 3.3.1 Das FUTURE Echtzeitsystem

Die Interprozessorkommunikation ist als Write-Only-Broadcastsystem in Lichtleitertechnik ausgeführt (vollständige Vernetzung); die I/O-Busse sind serielle Leitungen. Die Interprozessor-Nachrichtenpuffer sind im Adressraum ähnlich wie in ATTEMPTO organisiert (s. Abb. 2.3.10). Sie werden wie Speicherzellen adressiert und transparent in einem Schreibzyklus (3µs) mit dem Broadcastsystem aktualisiert.

Softwarekonfiguration

Für jeden periodischen Task wird statisch vom Benutzer festgelegt, welcher Fehlertoleranzklasse er zuzurechnen ist. Zur Auswahl stehen die Klassen 1 (normale Programmausführung), 2

(Fehlererkennung durch Vergleich der Ergebnisse zwei gleicher Prozesse) und 3 (TMR-Fehlermaskierung, s. Abschnitt 1.4.1), die einen, zwei oder drei Prozessoren für die Ausführung des Tasks benötigen [FÄR]. Diese Klassifizierung wird im Taskheader mitgeführt und bewirkt, transparent für den Task, einen Vergleich der Daten beim I/O und Interprozessor Datenaustausch. Alle für die Fehlertoleranz nötigen Prozeduren zum Scheduling, I/O Prozeduren, Nachrichtenüberwachung und Voting werden als Laufzeitbibliothek an die Anwendersoftware angebunden.

Fehlertoleranzmechanismen
Außer den Klassen 1 (nicht-fehlertolerant), 2 (Fehlererkennung und Vermeidung defekter Ausgabe) und 3 (Majoritätsentscheid) gibt es im Gegensatz zu ATTEMPTO keine höheren Fehlerklassen, die mehrere Simultanfehler zulassen.

Besondere Sorgfalt erfuhr das I/O-System. Jede I/O-Einheit führt periodisch einen Selbsttest durch und kann bei Übereinstimmung von mehr als zwei Verarbeitungsrechnern durch spezielle Leitungen extern abgeschaltet werden. Die Daten des I/O-Systems werden vor der Verwendung aufbereitet und dann in einem Puffer ("process image") zwischengelagert, aus dem sie von den Anwendertasks ausgelesen werden. Die Aufbereitung folgt dabei unterschiedlichen Algorithmen:

- *Zeitabgleich (Digitale Eingabe)*

  Ein Zustand eines von einem Datenbus (paralleler I/O) eingelesenen Signals (digitaler Wert) innerhalb einer Folge von digitalen Werten kann durch winzige Zeitverschiebungen in dem einem Sensoreingang bereits wahrgenommen werden, im anderen dagegen noch nicht.
  Zum Abgleich müssen die Serien der verschiedenen Eingänge innerhalb eines Zeitfensters miteinander verglichen werden, um herauszufinden, ob ein Sensor defekt ist oder nur eine Zeitverschiebung vorliegt.

- *Werteabgleich (Analoge Eingabe)*

  Um die Schwankungen bei der Eingabe analoger Signale (unterschiedliche Offset-Spannungen etc.) auszugleichen, muß eine potentielle Schwankungsbreite ("Wertefenster") definiert werden, innerhalb deren Grenzen alle verschiedenen Eingangswerte als gleich (nicht fehlerhaft) angesehen werden und für die weitere Verwendung interaktiv in allen Rechnern egalisiert werden.

Zum Test der Eingabesensoren werden periodisch Tests durchgeführt, bei denen von der Ausgabe künstlich generierte Signale bei der Eingabe eingespeist werden.
Ähnlich wie in ATTEMPTO überwachen die anderen Rechner bei den Klasse 2 und 3 Tasks die Ausgabe einer Ausgabeeinheit.

3.3.2 Einhalten von Zeitschranken

Das zweite Problem besteht in der Tatsache, daß es für Probleme der Prozeßkontrolle wichtig ist, bestimmte Aufgaben in einer bestimmten Zeit zu erledigen. Die garantierte Einhaltung von Zeitschranken ist deshalb ein besonderes Problem, das auch besonderer Behandlung bedarf.

Da die Zuteilung der Prozessorzeit üblicherweise transparent für das Benutzerprogramm vom Betriebssystem erledigt wird, stellt sich die Frage: welche Eigenschaften muß ein Betriebssystem haben, um Real-Time- Fähig zu sein?
Im Allgemeinen gehört beispielsweise das in ATTEMPTO verwendete UNIX nicht dazu, da folgende Echtzeitanforderungen nicht erfüllt werden:

- Die Prozeßwarteschlange (*run queue*) im Betriebssystemkern wird nach dynamischen Prioritäten (feste Prio und bisherige Laufzeit) geordnet (damit ist der UNIX-Scheduler non-deterministisch!); auch privilegierte Prozesse (negative Prio) haben keine festen Prioritäten und verlieren während ihrer Zeitscheibe ihren Vorrang. Aber auch hier wird nicht sofort auf den höchst-prioritären Prozeß umgeschaltet: Jeder Prozeß kann nur dann beendet werden, wenn er im User-Mode ist, seine Zeitscheibe abläuft (max. 16,6 ms Wartezeit) oder er sich freiwillig schlafen legt.
- Wird ein neuer Prozeß erzeugt oder werden die Speicheranforderungen eines existierenden Prozesses erhöht, so wird Speicher nicht dynamisch alloziert, sondern der Prozeß wird auf die Platte verlagert und als Ganzes wieder hereingeholt (*swapping*). Dieser Vorgang braucht vom Echtzeit-Standpunkt aus enorm viel Zeit.

Es gibt verschiedene Ansätze, die genannten Probleme durch Änderungen des UNIX-Kerns und der Treiber zu lösen, um garantierte Antwortzeiten bzw. ein deterministisches Systemverhalten zu erzielen. Beispielsweise ist in UNIX System V Version 4 die Schedulingstrategie nicht mehr fest eingebaut, sondern benutzerdefinierbar, um so allen Anforderungen gerecht werden zu können.
Eines der Systeme, die Echtzeitfähigkeit auch bei voller Belastung garantieren wollen, ist das MARS System.

MARS

Das **MA**intainable **R**eal-time **S**ystem (MARS) wurde an der Technischen Universität Wien konzipiert und stellt ein verteiltes System dar, in dem Anforderungen an Realtime-Fähigkeit und Fehlertoleranz besonders berücksichtigt wurden [KOP1],[KOP2].

Hardwarearchitektur

Das MARS-System besteht aus einem konventionellen Host-Entwicklungsrechner, der an ein Netzwerk von echtzeitfähigen Rechensystemclustern angeschlossen ist. In Abbildung 3.3.2 ist ein Überblick gegeben.
Eines der wichtigsten Ziele des MARS-Systems ist die Garantie der Echtzeitfähigkeit des verteilten Gesamtsystems. Dazu wird mit Hilfe eines fehlertoleranten Uhrensynchronisationsmechanismus und einer speziellen integrierten Schaltung im gesamten Cluster die Zeitbasis auf 10µs genau gehalten, die durch Funkuhrempfang mit der internationalen Atomzeit auf 100µs abgestimmt wird. Die Hardwareredundanz beschränkt sich im wesentlichen auf parallel arbeitende Cluster sowie der Einheiten innerhalb eines Clusters. Die Fehlertoleranzmechanismen sind in Software realisiert.

Softwarekonfiguration

Jedes Rechensystem innerhalb eines Clusters besitzt seine eigene Kopie des Betriebssystemkerns, der unter anderem die Nachrichtenübertragung und das Real-Time Scheduling implementiert.
Um den harten Echtzeitforderungen zu genügen, wird in jedem Rechner nur derjenige Timer-Interrupt berücksichtigt, der den Scheduler aufruft. Alle anderen Interrupts der I/O Komponenten werden gesperrt und nur im Zyklus einer Datenerfassung und -verarbeitung ("Datentransaktion") durch "polling" erfaßt. Damit ist die Prozessorzuteilung von äußeren I/O-Interrupts unabhängig und streng deterministisch und es kann die Einhaltung einer Prozessorzuteilung für jeden Task garantiert werden, sofern die maximale Verarbeitungszeit der Anwendertasks gegeben ist und die Rechnerkapazität es erlaubt.

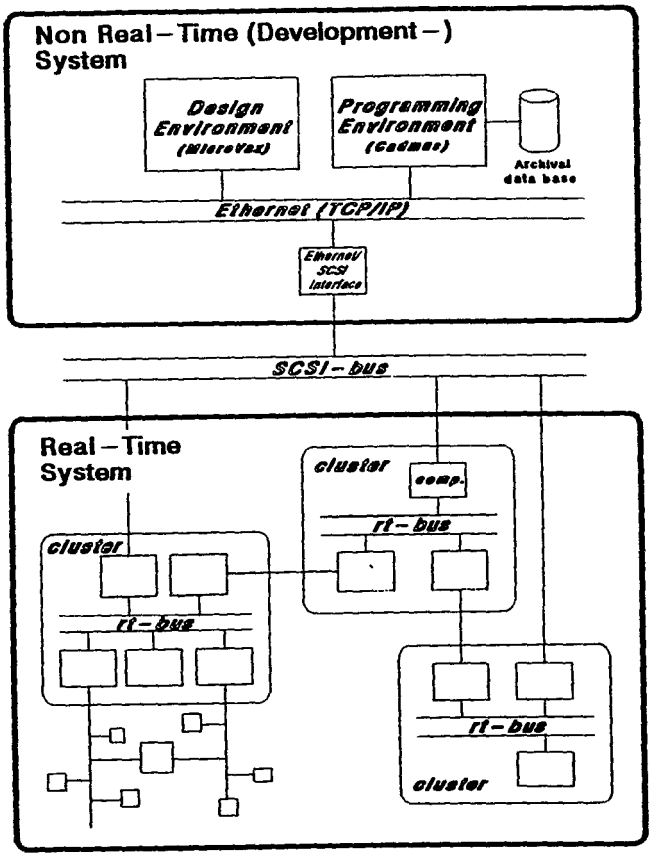


Abb. 3.3.2 Das MARS Hardwarekonzept (aus [KOP1])

Da beides meist nicht von vornherein für den Anwender überschaubar ist, wird in MARS ein interaktives Werkzeug MART zur Verfügung gestellt, mit dem die Laufzeitabschätzung und damit die CPU-Auslastung optimiert werden kann. Dazu wird für jeden Subtask eine Spezifikation sowohl der Funktion als auch der Zeitbedingungen ("contract") erstellt und die Konsistenz aller "contracts" geprüft.

Die Forderungen nach deterministischen, zeitbegrenzten Mechanismen werden auch beim Nachrichtensystem berücksichtigt. Alle Daten im MARS System werden in einem einheitlichen Format von einheitlicher Länge (1kB) transportiert. Innerhalb von MARS wird dazu nur ein Datagramm-Service angeboten, also ein ungesicherter Nachrichtentransport entsprechend den OSI-Schichten 1-3 (s. Abschnitt 1.4.2). Jede Nachricht wird zweimal übertragen, so daß die Fehlerrate von zwei unabhängigen, transienten Fehlern zweier aufeinander folgender Nachrichten beim verwendeten Ethernet mit 10^{-10} im Bereich der Hardwareausfallraten liegt. Da Echtzeitanforderungen meist auch mit zyklischen Tasks verbunden sind, reicht es hier aus, die Nachrichtenpuffer (z.B. Eingabedaten) bei den Empfängern immer nur mit den neuesten, taktsynchronen Nachrichten zu überschreiben; der Empfang und die Quittierung jeder gesendeten Nachricht ist nicht nötig; eine verspätete Nachricht ist in diesem System problematischer als gar keine Nachricht. In MARS können deshalb die oberen, zeitintensiven ISO-Protokollschichten weggelassen werden, ohne zu Dateninkonsistenzen zu führen.

Für Tasks, die keinen Echzeitanforderungen unterliegen (z.B. UNIX-Prozesse), existiert in MARS ein UNIX-Emulator, der als "lowest priority task" diese UNIX-Prozesse ablaufen läßt.

Fehlertoleranzmaßnahmen

Im MARS-System wird versucht, durch Konsistenztests (Datenformate, Doppelberechnungen, Zeitschranken etc) und Selbsttests Fehler zu entdecken und eine Rekonfiguration zu veranlassen. Die Überdeckung dieser konventionellen Selbsttests ist nach den letzten Untersuchungen relativ hoch [DAM].

Innerhalb der gesamten MARS Design Tools (MARDS) existiert auch ein spezielles Werkzeug MARPLE (*MA*rs *R*eliability *P*redictor and *L*ow-cost *E*stimator), mit dem die Struktur eines Computersystems bereits im Designstadium mit einer Abhängigkeitsanalyse überprüft werden kann. Dazu wird das System in einer speziellen Sprache spezifiziert, die als Sprachelemente unter anderem "Objekte", "Informationsdaten" und "Transaktionen" enthält und durch den MARPLE-Compiler in ein Abhängigkeitsmodell umgesetzt wird. Das generierte Modell kann anschließend zur Überprüfung des Systemdesigns auf Schwachstellen bei Komponentenausfall durch andere Werkzeuge, beispielsweise SHARPE [SAHN], analysiert und dargestellt werden.

3.4 Das ATTEMPTO System

Im ATTEMPTO System wird versucht, dem Benutzer korrekte Ergebnisse seines Programms mit Hilfe von softwareimplementierten Fehlertoleranzmechanismen zu garantieren.

Im Abschnitt 2.3 wurde bereits ein Überblick über die Hardwarestruktur sowie die Mechanismen des dezentralen Betriebssystems von ATTEMPTO gegeben. Zusätzlich sollen nun in diesem Abschnitt genauer auf die Konzepte und Mechanismen der Fehlertoleranz eingegangen werden.

Beginnen wir mit den zusätzlichen Anforderungen. Zu den in Abschnitt 2.3.1 festgestellten Mängeln vieler bisheriger Systeme, wie anwenderimplementierte Fehlertoleranzmechanismen (z.B.Tandem!) oder teure Spezialhardware für Fehlertoleranzzwecke (z.B.FTMP), muß noch ein weiteres wichtiges Problem erwähnt werden:

- Die Fehlertoleranz umfasst meist nur Teile des Gesamtsystems

In den folgenden Abschnitten sollen nun die Fehlertoleranz-Anforderungen und Mechanismen beschrieben werden.

3.4.1 Fehlertoleranz-Anforderungen

Speziell für Fehlertoleranz ergeben sich zusätzliche Forderungen zu denjenigen aus Abschnitt 2.3.1 an das System:

6) Die Fehlertoleranzmechanismen müssen modular und hardwareunabhängig gewählt werden.

7) Alle Fehlertoleranzmechanismen wie Fehlerdiagnose, Fehlermaskierung oder Rekonfiguration müssen völlig dezentral funktionieren, um trotz Ausfalls irgendeiner Komponente ein Weiterfunktionieren des Systems zu garantieren.

8) Die Fehlertoleranz erstreckt sich bis auf die Ein- und Ausgabeleitungen.

3.4.2 Konzepte der Fehlertoleranz

Für die softwareimplementierte Fehlertoleranz kommen, wie in Abschnitt 3.2 gezeigt, verschiedene Konzepte in Betracht. Für das ATTEMPTO System entwickelten wir zwei verschiedene Konzepte, die verschiede Fehlertoleranz-Anforderungen repräsentieren und modular aufeinander aufgebaut sind. Betrachten wir zunächst das aufwendigere Maskierungskonzept von ATTEMPTO 1, das mit dem von SIFT (Abschnitt 3.2) vergleichbar ist.

Das Fehlertoleranzkonzept von ATTEMPTO 1

Das Test- und Diagnosekonzept von ATTEMPTO 1 basiert im wesentlichen auf einer End-to-End Strategie, bei der nur die Korrektheit der zur Ausgabe bestimmten Ergebnisse durch einen software-

implementierten Fehlertoleranzmechanismus garantiert werden soll. Bei einem Ausfall auf einem SBC führen alle lokalen Funktionen, da es sich um nicht-redundante, konventionelle Hardware handelt, zu diversen lokalen Fehlern, die aber trotz ihrer Vielfalt nur zwei verschiedene Auswirkungen in diesem Modell haben können: die Ergebnisse bleiben korrekt oder werden falsch.

Da wir eine funktionsbegleitende Fehlermaskierung in dem System wünschten, kamen für uns als Test- und Diagnosemethoden hauptsächlich nur Mehrheitsentscheidungen (*majority voting*, s. Kapitel 1.4) und Vergleich der Rechenergebnisse (*Vergleichstestverfahren*, s. Kapitel 1.5.1) in Betracht. Vergleichen wir kurz diese beiden Diagnoseverfahren bezüglich ihres Aufwands und ihrer Möglichkeiten.

Test und Diagnose

Wenn wir jeden Single-Board Computer (SBC) mit einer aktiv testenden Einheit identifizieren, so ist das Vergleichstestverfahren mit maximal t = N-2 diagnostizierbaren defekten Einheiten der Mehrheitsentscheidung mit maximal t = $\lfloor (N-1)/2 \rfloor$ ebenbürtig (N=3) oder überlegen (N>3). Betrachten wir außerdem die Zahl der Testverbindungen für die Zahl der zu sendenden Nachrichten, so benötigt das Vergleichstestverfahren $|E| \geq \lceil N\ t/2 \rceil$ Testverbindungen, die Mehrheitsentscheidung dagegen N(N-1) Testverbindungen. Aus diesen beiden Gründen entschieden wir uns für das Vergleichstestverfahren als Diagnosealgorithmus in ATTEMPTO 1 [AM1]. Verwendet man zusätzliche, diagnoseabhängige Testprotokolle, so läßt sich die Zahl der Nachrichten im Vergleichstestmodell weiter reduzieren [DAL5].

Fehlertoleranz wird dadurch erreicht, daß derselbe Benutzerjob auf mehreren SBC ausgeführt wird und hinterher die Ausgaben verglichen werden (Fehlermaskierung). Danach werden die durch Vergleichstest ermittelte, korrekte Ausgabe von einem SBC unter Kontrolle der anderen ausgegeben.

Vermeidung von fehlerhafter Ausgabe

Vor jeder Ausgabeoperation vergleichen alle Prozessoren, die dasselbe Programm bearbeiten, die zur Ausgabe anstehenden Daten. Jedem Fehlertoleranzgrad ist ein Testgraph von Vergleichstests zugeordnet, durch den festgelegt ist, welche Ausgaben miteinander zu vergleichen sind.
Ist eine Ausgabe vom UserJob erstellt und zur FTL weitergeleitet worden, so wird von den Ausgabedaten die Signatur gebildet und an alle beteiligten Kollegen geschickt. Jeder Kollege vergleicht nach Erhalt aller Signaturen (Time-Out!) diese miteinander.
Die Diagnose basiert auf den Ergebnissen dieser Vergleiche. Damit gewinnt jeder SBC Information über den Zustand der Kollegen. Die eigentliche Ausgabeoperation wird daraufhin von dem Prozessor durchgeführt, der einen Ausgabeschlüssel angefordert und erhalten hat und darauf in der Bewerbung als "IOMaster" als erster auftritt. Dann bewirbt er sich normal um die globale Resource (z.B. das Terminal) und benutzt sie. Die anderen intakten Kollegen überwachen diese Ausgabe (Echo,s.Abschnitt 2.3.2).

Da wir primär transiente Fehler annehmen, ist es nicht ratsam, bei jedem auftretenden Fehler den SBC auszutauschen. Stattdessen führt jeder SBC eine eigene Fehlerfrequenzliste wie in C.mmp (s. Abschnitt 2.1), in der auftretende Nichtübereinstimmungen der Resultate für spätere Auswertungen durch ein globales Abstimmungsverfahren, funktional ähnlich wie in SIFT (s. Abschnitt 3.2) notiert werden. Direktes Resultat eines nichtübereinstimmenden Vergleichs ist ein Selbsttest der betreffenden Einheit, um auftretende Ausfälle möglichst früh diagnostizieren zu können und im

Fall eines intransienten Fehlers den betreffenden SBC zu veranlassen, keine weiteren Job-Bewerbungen mehr durchzuführen.

Das Fehlertoleranzkonzept von ATTEMPTO 2

Das Fehlertoleranzkonzept in ATTEMPTO 1 baut im Wesentlichen auf einer Replizierung der auszuführenden Tasks und anschließender dezentralen Fehlermaskierung und Voting auf. Dabei ist die Größe einer zu replizierenden Task identisch mit dem auszuführenden Auftrag; mit dem Job als kleinster, zu replizierenden Einheit ist die Granularität der Fehlertoleranz ziemlich grob. In Abbildung 3.4.1 ist als Beispiel eine Kommunikationsstruktur für TMR gezeigt.

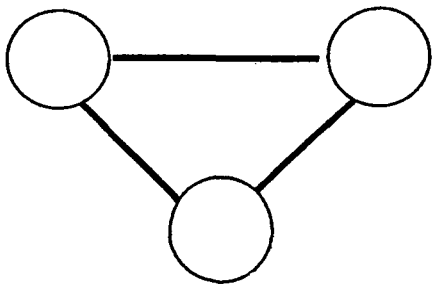


Abb. 3.4.1 Eine fehlertolerante TMR-Struktur in ATTEMPTO I

Demgegenüber sieht das Arbeitsmodell der Parallelisierung im ATTEMPTO 2 - System eine relativ feinkörnige Aufteilung der Gesamtarbeit vor (s. Abschnitt 2.3.5). Ausgehend von einem Hauptprogramm (*Wurzel*) werden parallel verteilbare Arbeitsteile auf anderen Prozessen anderer Prozessoren (*Unterknoten*) mit dem Modell des *Remote Procedure Call (RPC)* abgearbeitet. Diese Unterknoten können nun ebenfalls ihrerseits weitere Unterknoten mit parallel verteilbaren Teilen ihrer Teilarbeit beauftragen, so daß insgesamt die Auftragsstruktur einen Baum bildet. In Abbildung 3.4.2 ist ein solches Beispiel gezeigt.

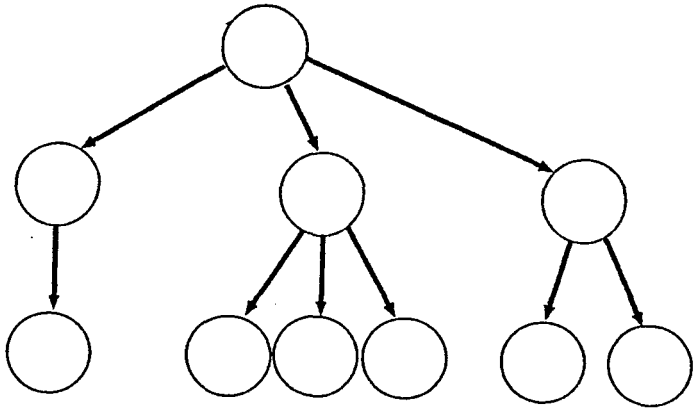


Abb. 3.4.2 Modell der Auftragsvergabe in ATTEMPTO 2

Wie könnte nun ein Fehlertoleranzmodell, besonders hinsichtlich eines Multiprozessorsystems mit vielen Prozessoren (z.B. Transputer-Netz), aussehen ? Dabei müssen wir folgende Fakten beachten:

- Eine volle Replikation ist in vielen Systemen aufgrund von Speicherbeschränkungen nicht

möglich. Beispielsweise haben Transputersysteme mit vielen Transputern meist wenig Speicher pro Transputer (ca 1MB) und keine Möglichkeit, elegant Prozeßteile (z.B. durch swapping oder paging) auf externe Massenspeicher auszulagern.

- Ausfälle passieren nicht so häufig im System, daß man immer nur im fehlermaskierenden Betrieb (ATTEMPTO 1) arbeiten müßte.

- Andererseits sollte schon gewährleistet sein, daß nach langer Arbeitszeit (z.B. langwierigen Berechnungen) nach einem Auftreten (Fail-save!) und Diagnose eines Hardware-Fehlers ("gutartige, eingrenzbare Fehler") der Job mit den letzten Teilergebnissen neu aufgesetzt und weiterrechnen kann.

Aus diesen drei Annahmen lassen sich folgende drei Schlußfolgerungen ziehen:

1) In ATTEMPTO 2 sollte es möglich sein, unterschiedliche Prozeßcode-Pakete zu schnüren, die nur den zur Ausführung eines Auftrags nötigen Programmcode enthalten. Das Konzept, auf alle Prozessoren das volle, alle Code-Teile aller nur möglichen Aufträge enthaltene Programm (*E-Prozeß-Konzept*) zu kopieren, ist bei beschränktem Speicher nicht ratsam.

2) Eine Replikation eines Auftragnehmers für Fehlertoleranzzwecke findet nicht statt. Stirbt der Auftragnehmer (Fail-Save!), so beauftragt der Auftraggeber (*Manager*) nach Test und Diagnose einen neuen Prozessor mit den alten Daten.

3) Nur die obersten Auftraggeber sind repliziert und bilden eine Oligarchie.

Das Konzept aus 2) und 3) läßt sich durch eine Kombination von Abbildung 3.4.1 und 3.4.2 verdeutlichen; die sind in Abbildung 3.4.3 gezeigt.

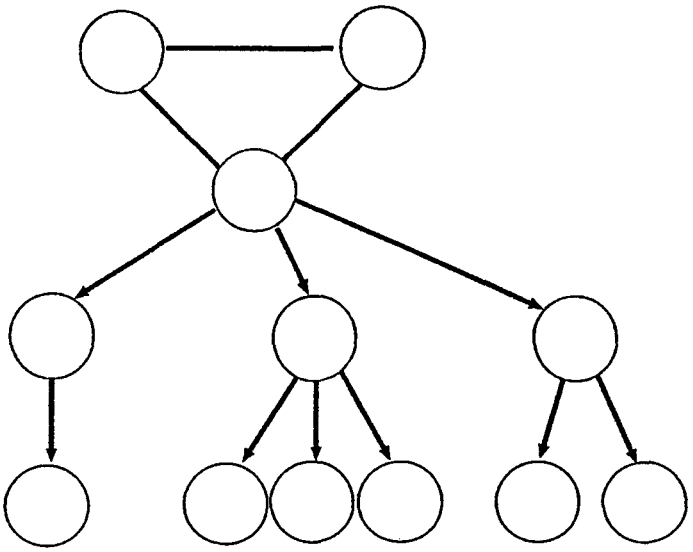


Abb. 3.4.3 Fehlertolerante, hierarchische Auftragsstruktur

Eine andere Möglichkeit besteht darin, die Aufträge unter den Einheiten der fehlertoleranten Auftraggeber-Oligarchie aufzuteilen, die wiederum jede für sich Untereinheiten mit Unteraufträgen

beschäftigen. In Abbildung 3.4.4 ist eine solche Auftragsstruktur zu sehen.

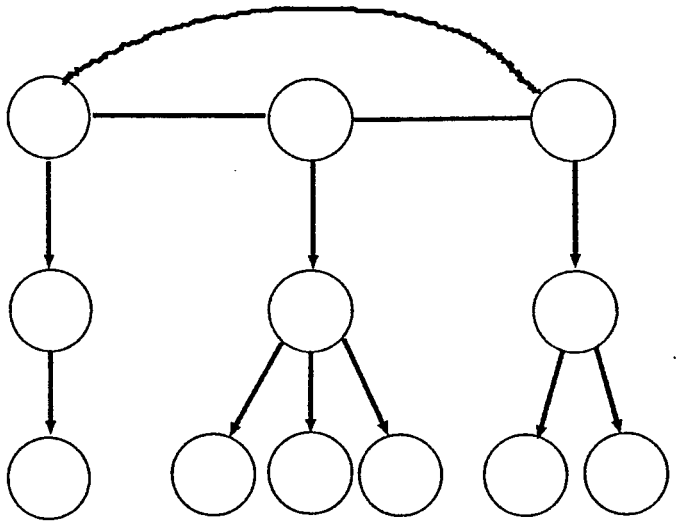


Abb. 3.4.4 Auftragsverteilte Oligarchie

Betrachten wir das Auftrags- und Fehlertoleranzkonzept 2) und 3), dessen Struktur in Bild 3.4.4 angedeutet ist, so gibt es dafür bereits ein Vorbild in der Literatur. In [WITT] stellten A. van Tilborg und D.Wittie für ihr MICROS-System die Idee einer Fehlertoleranz-Oligarchie und in [TIL] ein Auftrags-Konzept unter dem Namen "Wave-Scheduling" vor. Dem liegt die Vorstellung zugrunde, daß die Aufträge "wie eine Wellenfront" einheitlich durch die Ebenen des Baumes bis in die "entferntesten" Prozessoren (Blätter) vordringen und die Ergebnisse ebenso "wie eine reflektierte Welle" zurück zum Auftraggeber gelangen.

Mit den vier Voraussetzungen

1) Die Prozessoren sind alle gleich (*Homogenität*)
2) Die Aufträge sind einheitlich (gleiche Länge, *unit tasks*); jeder Prozessor (Knoten) kann nur eine ganze Zahl von tasks gleichzeitig ausführen
3) Der Scheduler weiß im voraus nicht,
   - wieviel tasks bei einem schedule zu bewältigen sind
   - was jede task an resourcen benötigt
   - wie groß die Inter-task Kommunikation zwischen zwei tasks ist
4) Die Betriebssystem-Auftragsstruktur (Scheduler, Dispatcher) entspricht der Abbildung 3.2.4. Die Blätter des Baums sind Arbeiter, alle anderen Knoten Manager, die die Verteilung und lokale Resourcenbeschaffung übernehmen.

zeigten die Autoren, daß das verteilte Wave-Scheduling-Verfahren Resultate (Auslastung) bringt, die denen eines zentralen Scheduling vergleichbar sind. Dabei wird auch (im Unterschied z.B. zum Diffusions-Scheduling) ein statischer Deadlock vermieden.

Da die auftragsverteilte Oligarchie mit ihren multiplen Auftraggebern auf oberster Stufe sich schwer in das Parallelisierungskonzept (s. Kapitel 2.3.5) mit einem einzigen Auftraggeber als Wurzel eines Baumes (*farming-Modell*) einfügt, wird stattdessen die fehlertolerante, homogene Auftragsstruktur mit repliziertem Auftraggeber aus Abbildung 3.4.3 für das Fehlertoleranzkonzept von ATTEMPTO 2 gewählt.

Das Grundkonzept enthät dabei zwei unterschiedliche Fehlertoleranzmechanismen. Für die oberste Oligarchie gilt das Konzept von ATTEMPTO 1, also eine Redundanz in der Arbeitsausführung und anschließender Diagnose für das Ergebnis. Durch eine Kennzeichnung der Auftragsvergabe als "globale Resource" wird automatisch im System der Vergleichsprozeß angestoßen. Die Auftraggeber überprüfen sich also über den Signaturvergleich der Auftragsnachrichten. Die Ergebnisse müssen anschließend von den beauftragten Prozessoren als Nachrichten in einem "Broadcast" Modus an den replizierten Auftraggeber zurückgegeben werden.

Alle Unteraufträge (Auftragsverzweigungen) bekommen andere Fehlertoleranzmechanismen. Da die Speicherung von Zwischenergebnissen durch die Oligarchie gesichert ist, muß für die Ausführung der eigentlichen Arbeit nicht mehr hohe, fehlermaskierende Redundanz vorgesehen werden. Stattdessen reicht es aus, mit den Fehlertoleranzforderungen 2) und 3) systematisch Rücksetzpunkte bei der Auftragsvergabe zu schaffen. Dazu bietet sich der Mechanismus der Auftragsvergabe geradezu an: bei der Auftragsvergabe werden alle Informationen in Form von Datenstrukturen und Puffern aufgehoben. Werden die Ergebnisse vom beauftragten Prozessor zurückgeschickt, so werden die alten Funktionsparameter mit den neuen Werten überschrieben und der Rücksprung in die aufrufende Prozedur eingeleitet (s. Abschnitt 2.3.5). Bemerkt dagegen der Auftraggeber durch Zeitüberwachung (Time-Out) des Auftrags oder durch ausbleibende Kontrollnachrichten (Totmann-Schalter oder Heart-beat) den Abbruch des Auftrags, so sind alle Informationen noch vorhanden, um ihn erneut aufzusetzen.

Allerdings verlangt ein solches Konzept, daß die Auftragnehmer-Prozessoren Fail-save Eigenschaft (s. Kapitel 1.4) haben. Dies ist durch die üblichen Fehlererkennungsmechanismen, wie Selbsttests, Floationg-Point Error, Memory Parity Check etc. nicht immer gewährleistet (vgl. z.B. [DAM]). Will man eine verläßliche Fehlertoleranz auch bei funktionellen Fehlern, so kommt man um eine Funktionsüberwachung, beispielsweise durch einen zweiten Prozessor, nicht herum. Um die Fail-save Eigenschaft aller beteiligten Prozessoren sicherzustellen, ist es also nötig, jeweils die Auftragsvergabe mindestens zu duplizieren. Im fehlertoleranten Betrieb vergibt jeder Auftraggeber zwei Aufträge und vergleicht anschließend die Ergebnisse (Vergleichstest, s. Abschnitt 1.5.1) miteinander. Besteht ein Unterschied, so muß der Auftrag erneut vergeben werden, anderfalls werden die Ergebnisse akzeptiert.

Das Fehlertoleranzkonzept von ATTEMPTO 2 benutzt also im Wesentlichen *roll-back*, ähnlich wie im Tandem-System (s. Abschnitt 3.2), wobei die Daten der Rücksetzpunkte durch die Auftragspufferung im Auftraggeber-Leichtgewichtsprozeß des RPC (s. Parallelisierungskonzept, Abschnitt 2.3.5) automatisch und benutzertransparent vorhanden sind.

3.4.3 Implementierung

Für das konkrete Implementierungskonzept von ATTEMTO 1 wirken sich die Vorteile des Diagnoseverfahrens nicht besonders stark aus. Dem liegt Folgendes zu Grunde:

- Die logisch getrennten Nachrichtenwege liegen physikalisch auf *einem* Bus; die Nachrichten werden im Broadcast-Modus (s. Kapitel 3.3.4) verschickt. Damit kann prinzipiell jede Einheit ohne Mehraufwand die Nachrichten empfangen. Da beim Vergleichstestmodell jede Einheit ihr Ergebnis mindestens einem Nachbarn (*Kollegen*) mitteilt, wird durch die

Broadcast-Eigenschaft der Nachrichtenübermittlung in N Nachrichten prinzipiell jedes Ergebnis jeder Einheit übermittelt, so wie dies auch für die Mehrheitsentscheidung nötig ist. Das zusätzliche Auslesen und Weiterbearbeiten der Nachrichten bei der Mehrheitsentscheidung wird durch den relativ einfachen Algorithmus mehr als kompensiert.

- Die hohe Zahl von defekten Einheiten, die beim Vergleichstestverfahren gegenüber dem Mehrheitsverfahren diagnostiziert werden können, beruht hauptsächlich auf der Annahme, daß keine zwei defekten Ergebnisse gleich sein können. Gehen wir von der Realität aus, daß eine gemeinsame Ursache bei mehreren Einheiten die gleiche Wirkung hervorbringen kann[1)], so müssen wir auch einen latenten systematischen Fertigungsfehler, beispielsweise in den Chip-Fertigungsmasken, oder einen gemeinsamen transienten Fehler für mehrere SBC (Massepotential-Störung, Transienten im Starkstromnetz u.ä.) in unserem Fehlermodell berücksichtigen. Da die Computerboard-Funktionen nicht streng synchronisiert, sondern nur koordiniert sind, wird ein gemeinsamer transienter Fehler auf verschiedenen Boards verschiedene Softwareteile betreffen. Da man aber nicht ausschließen kann, daß bei mehreren SBC die gleichen Software-Module oder die gleichen Chips sich als Schwachstellen erweisen werden, ist das Vergleichstestmodell, das gemeinsame Fehler bei zwei SBCs nicht vorsieht, nur durch die geringe Auftrittswahrscheinlichkeit derartiger kritischer Fehler zu rechtfertigen. Hierbei fehlt allerdings bisher verläßliches, statistisches Material.

  Allerdings gilt diese Kritik auch für NMR-Systeme, falls eine Mehrheit der Prozessoren von einem gemeinsamen Fehler betroffen ist. Besteht diese Mehrheit aus zwei Prozessoren wie in TMR-Systemen, so zeigen beide, Vergleichstestverfahren und Majoritätsentscheidung, fehlerhafte Ergebnisse. In der jetzigen ATTEMPTO 1- Implementierung mit drei SBCs ist damit keines der beiden Verfahren bezüglich potentieller Simultanfehler besser geeignet.

Aus den beiden genannten Gründen ist der Diagnosealgorithmus in ATTEMPTO 1 als eigenständiges Modul "SAB" realisiert und kann leicht ausgetauscht werden, wenn der Einsatz der Vergleichstests sich in der Praxis als problematisch erweisen sollte.

Dabei sollte man aber beachten, daß die Bezeichnungen "Vergleichstestverfahren" und "Mehrheitsentscheidung" nur das Grundverfahren beschreiben, nicht aber die Implementierung. Beispielsweise benötigt eine Mehrheitsentscheidung eine (ultrazuverlässige) Entscheidungseinheit. Durch die Replizierung dieser Entscheidung auf jedem SBC und dem anschließenden Ausgabeprotokoll wird die Funktion der ultrazuverlässigen Entscheidungseinheit fehlertolerant den SBCs übertragen, so daß das klassiche Mehrheitsentscheidungsmodell in dieser Form gar nicht vorliegt.

In der folgenden Abbildung zeigt eine Beschreibung des Signatur-Verwaltungsmoduls den genaueren Ablauf der Nachrichtenfolge.

MODULE SAB
Datenstruktur: Signature Array Buffer SAB : Liste von Verbunden mit Job-Namen, Nummer des Outputs, Signatur der Output-Daten usw.
aktive Einheit: SABclerk, verwaltet SAB, veranlaßt Vergleich bzw. Diagnose
Botschaften:
OutputData: Bildung der Signatur der empfangenen Output-Daten, Versenden dieser Signatur an alle Kollegen-SABclerks (aus Gründen der Synchronisation auch an sich selbst) mit Kennung 'Signature', zugleich Aufsetzen eines Timeout
Signature: Eintragen der empfangenen Signatur in den SA-Puffer. Sobald alle Signaturen vorhanden, Vergleich und Diagnose. Ausbleibende Signaturen gelten als fehlerhaft. Danach Versenden einer Botschaft mit Kennung *'DemandKey'* an bestimmten Kollegen-SABclerk (KeyPartner), zugleich Aufsetzen eines

[1)] Beispielsweise wurden einmal im Atomkraftwerk Grohnde an der Weser in vier parallel geschalteten Pumpen gleichzeitig aus gleicher Ursache die gleichen Störungen ausgelöst [Frankfurter Rundschau, 25.9.89, S.4] .

Systemstruktur (UNIX- und Modula-2-Prozesse) stark modularisiert.

Deshalb wurde für die Testphase die gesamte Software in Einzelmodule zerlegt und die Reaktionen auf alle spezifizierten Parameterwerte (z.B. Nachrichtentypen) einzeln getestet. Da es sich bei der Simulation um ein nur begrenzt zeitkritisches System (nur Time-Out der Kommunikation) handelt, ist dies relativ gut durchzuführen.
Im Einzelnen handelt es sich dabei um folgende Module:

Zuerst wurde das Programm zum Erzeugen des Simulationsrahmens (s. Abschnitt 2.3.6) erstellt und ausgetestet, ob auch alle Prozesse und Pipes mit den richtigen, vorher vereinbarten file-Deskriptoren erzeugt wurden. Dann wurde der Code der UNIX-Prozesse, die danach den Prozessen im Simulationsrahmen überlagert werden, separat getestet.

Das gewünschte Input-Output-Verhalten der Prozesse TTY, TT, SC, SV, SH/UJ kann leicht durch Testnachrichten überprüft werden. Dieser Testinput wird vorher auf Dateien geschrieben; bei der Test-Aktivierung des Prozesses wird von der Möglichkeit der UNIX-Shell Gebrauch gemacht, den Testinput auf den Standard-Inputkanal 0 und den Testoutput auf einen Log-File umzulenken.

Da die Fehlertoleranzschicht FTL als einzelner UNIX-Prozeß in mehrere Modula-2 Prozesse unterteilt ist, die über Nachrichten und Briefkästen miteinander kommunizieren, kann durch kontrollierte Eingabe von Testnachrichten (Testfiles) sowie durch Protokollieren aller durch das Post-Office laufenden Nachrichten (s. Abb. 2.3.4) genau festgestellt werden, welcher Modula-2 Prozeß falsches Input-Output im Zusammenspiel mit den anderen liefert.

Das Prozeßmanagement, welches das von Modula-2 zur Verfügung gestellte Koroutinen-Konzept benutzt, behandelt die auf dem benutzten Rechner (PDP11/24) nötigen Overlays transparent für die Prozesse und konnte somit getrennt ausgetestet werden.

Um die Überprüfung der Datenbasis der FTL und des Laufzeitverhaltens der Modula-2 Prozesse auf Source-Code-Level zu erreichen, wurde der Post-Mortem Debugger von Modula-2 [MAI] für UNIX leicht modifiziert. Anstatt den zu überwachenden UNIX-Prozess direkt zu starten, wird zuerst der Debugger aufgesetzt, der wiederum als Kind-Prozeß den fraglichen Prozeß erzeugt. Empfängt der mit dem UNIX-SysCall *ptrace()* initialisierte Kind-Prozeß ein Signal (z.B. vom Debugger), so wird er als Prozeß "eingefroren" und die Kontrolle geht an den Debugger über. Zum eigentlichen Debuggen wird der Speicherinhalt nicht aus dem Post-Mortem-Dump file, sondern direkt mittels des UNIX SysCall "ptrace()" von dem angehaltenen, zu überwachenden Kind-Prozeß ausgelesen und in Source-Code Form (Datentypen Variablennamen, etc) dargestellt. Die Änderung, die den Post-Mortem-Debugger in einen Online-Debugger umwandelt, liegt in dem Ersatz der untersten Schicht des Debuggers, der Funktion "CoreWord()", die ein Datenwort vom Post-Mortem-Dump file liest, durch den UNIX-SysCall "ptrace()", der dies von einem Kindsprozeß on-line durchführt.

Verifikation imperativer Programme

Im Unterschied zu der Philosophie, jedes Programm als Kunstwerk zu betrachten und es mit ausgewählten Daten zu testen, zeigten die Bestrebungen der letzten Jahrzehnte, wie man dies besser systematisch für alle möglichen Daten durch ein logisches Kalkül verifizieren kann. Ausgehend von den axiomatischen Semantikdefinitionen von C.A.Hoare (Spezifikation von PASCAL 1971) über E.W.Dijkstra [DIJ2] zu D.Gries [GRI] zeigte sich, daß eine mechanische Verifikation auch größerer, blockorientierter Programme möglich ist.

Hierbei wird das Programm systematisch in seine Einzelblöcke zerlegt und für jeden

Einzelblock eine Verifikation durchgeführt. Jede der drei möglichen Blockstrukturen *Sequenz* (Anweisungen), *Alternative* (IF..THEN, CASE..) und *Iteration* (REPEAT..UNTIL.., WHILE..DO..), die selbst wieder Blöcke enthalten und damit auch Teil eines anderen Blocks sein können, wird durch ein bestimmtes Verfahren verifiziert. Dazu wird jeweils der Block durch eine Vorbedingung (*Precondition*) und eine Folgebedingung (*Postcondition*) über die verwendeten Parametervariablen ergänzt. Eine prädikatenlogische Spezifikation des Blocks beschreibt, wie die Precondition in die Postcondition überführt wird. Dabei wird nur spezifiziert, *was* getan werden soll, nicht *wie*; eine Spezifikation erlaubt deshalb meist viele Implementierungsmöglichkeiten.

Ausgehend von den größten Blöcken kann nun sukzessiv bis in die kleinsten Blöcke verifiziert werden, ob die Spezifikation tatsächlich erfüllt ist. Bei jeder kleineren Untereinheit werden die Preconditionen des Blocks nach bestimmten Regeln umgeformt und dort als Preconditionen verwendet. Entsprechend ergeben sich die Postconditionen des Blocks aus den Postconditionen der Unterblöcke.

Bei Prozeduren reicht es, die Prozedur selbst einmal zu verifizieren und für den Prozeduraufruf nur die aktuellen Pre- und Postconditionen zu erstellen. Allerdings müssen alle verwendeten, externen Variablen bei der Prozedurdeklaration als Parameter auftreten, da sonst unkontrollierbare Seiteneffekte (Globale Variable!) auftreten können. Dies entspricht in großem Maße auch den Anforderungen für verteilte Programmierung nachrichtengekoppelter Systeme, die beispielsweise im Parallelisierungskonzept von ATTEMPTO 2 (s. Abschnitt 2.3.5) Verwendung finden.

Auch parallele Programme in speichergekoppelten Systemen können verifiziert werden. Dazu muß im wesentlichen neben der Verfikation der sequentiellen Teile geprüft werden, ob nicht eine Interferenz mit anderen Prozessoren beim Zugriff auf Daten auftritt (z.B. globale Variable), s. [FUT]. Dies war beispielsweise bei der SIFT-Verifikation in Abschnitt 3.2 für den Scheduler nötig.

Obwohl das Verifikationsverfahren die Programmentwicklung selbst erleichtern kann (Pre-und Postcondition stehen dann im Programmkommentar) bietet sich eher an, große Programme mit mechanischen Hilfsmitteln (*verifikation tools*) zu verifizieren oder stattdessen eine ausführbare Spezifikation, beispielsweise in PROLOG, zu erstellen.

Verifikation des ATTEMPTO Systems

Ein genauer Nachweis der Fehlertoleranzeigenschaften des ATTEMPTO Systems läßt sich nicht, wie vorher erwähnt, nur auf erfolgreiches Testen und Debuggen stützen, obwohl dies für die Entwicklung und Implementierung sehr wichtig ist. Stattdessen muß von Schnittstelle zu Schnittstelle für jedes einzelne Modul logisch gezeigt werden, daß die beabsichtigte Wirkung auch eintritt. Wie soll dies für das ATTEMPTO System durchgeführt werden?

Ein wichtiges Beispiel einer Verifikation eines fehlertoleranten Multiprozessorsystems ist die mechanische Verifikation des SIFT Systems aus Abschnitt 3.2. Hier wurde zum ersten Mal versucht, systematisch das gesamte Design des Computersystems über die Verifikation der sequentiellen Programme hinaus zu verifizieren.

Zweifelsohne enthält dieser Ansatz viele interessante Ideen, die für die Verifikation des ATTEMPTO-Systems übernommen werden können. Beispielsweise ist die strukturelle Modularisierung sehr nützlich, bei der die Hauptmechanismen auf einer Ebene, einfach und klar spezifiziert, durch Subebenen detaillierter ausgeführt werden (s. Abb. 3.2.2). Auch die Zuverlässig-

keitsanalyse läßt sich so quantitativ mit den Mechanismen der Fehlertoleranz koppeln.
Allerdings gibt es auch Unterschiede, die zu beachten sind. Da das ATTEMPTO-System durch den unmodifizierten UNIX-Kern kein Real-Time-System (s. Abschnitt 3.3) sein kann, läßt sich der Begriff des Task-Zeitrahmens (execution window, data window) aus dem SIFT System hier nur sehr weit fassen. Im Wesentlichen müssen als Zeitbedingungen nur die Time-Out Zeiten der unteren Schichten (Zeitrahmen der Kommunikation, etc, s. Abschnitt 2.3.4) und der oberen FTL-Schichten (s. Abschnitt 2.3.3) beachtet werden.
Auch die SIFT-Notation der iterierten Tasks tritt hier nicht auf: jeder Task wird nur einmal ausgeführt; die Votierungszeitpunkte sind ereignis- und nicht zeitsynchronisiert.

Bestimmte Teile der verwendeten Soft-und Hardware können nicht verifiziert werden, da keine Informationen darüber vorliegen. Beispiele dafür sind der binäre Betriebssystemkern, Compiler, Mikroprozessor und diverse Hardwarekomponenten. Hier muß (leider!) auf die Korrektheit und Vollständigkeit der Spezifikation vertraut werden, die der Hersteller in Manuals zur Verfügung stellt und die für die Verifikation in zu erstellenden prädikatenlogischen Pre-und Postconditionen sowie Funktionsspezifikationen Niederschlag findet. In der Abbildung 3.4.6 sind die nichtverifizierbaren Teile schraffiert gezeichnet.

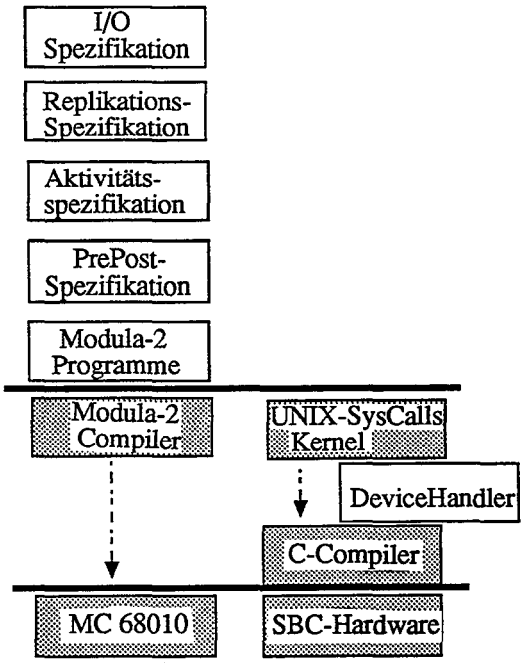


Abb. 3.4.6 Verifikationsschichtung in ATTEMPTO

Die Verifikation des SIFT-Systems beruht im Wesentlichen auf der Prädikatenlogik erster Ordnung, wobei die Modellierung und Verifikation des Systems durch ein spezielles Sprachsystem "STP" durchgeführt wird, das solche Konstrukte direkt unterstützt.
Andere Versuche, wie beispielsweise die Kit-Studie [BEV] zur Verifikation eines kleinen Betriebssystemkerns ohne Prozeßerzeugung und Filesystem, benutzen dazu auch Lisp-ähnliche, logische Konstrukte der Boyer-Moore Logik [BOY].
Für ATTEMPTO bietet sich die Möglichkeit an, Prädikatenlogik erster Stufe einzusetzen. Die verschiedenen Spezifikationsebenen lassen sich als Module in einem PROLOG-System formulieren, deren Korrektheitsbeweis mittels spezieller PROLOG-Klauseln geführt werden kann.

Im Unterschied zu STP lassen sich durch die Anbindung spezieller Prädikate an Betriebssystemfunktionen (z.B. in Modula-Prolog [MULL]) die Spezifikationen als ausführbare Programme betrachten, so daß sogar im Extremfall der existierende, explizite, imperative Programmcode in Modula-2 für die FTL nicht nötig wäre.

Petri-Netze

Eine andere Möglichkeit, parallele Systeme zu spezifizieren und zu verifizieren bietet die Modellierung mit Petri-Netzen. Sind die Petri-Netze einmal aufgestellt und formal gefaßt, so lassen sich Aussagen über Dead-Locks, Life-Locks und Invarianten des Netzes machen, die den formalen Nachweis der gewünschten Eigenschaften ermöglichen. In der folgenden Abbildung 3.4.7 ist eine solche Spezifikation des ATTEMPTO 1 Nachrichtenprotokolls der Interprozessorkommunikation gezeigt.

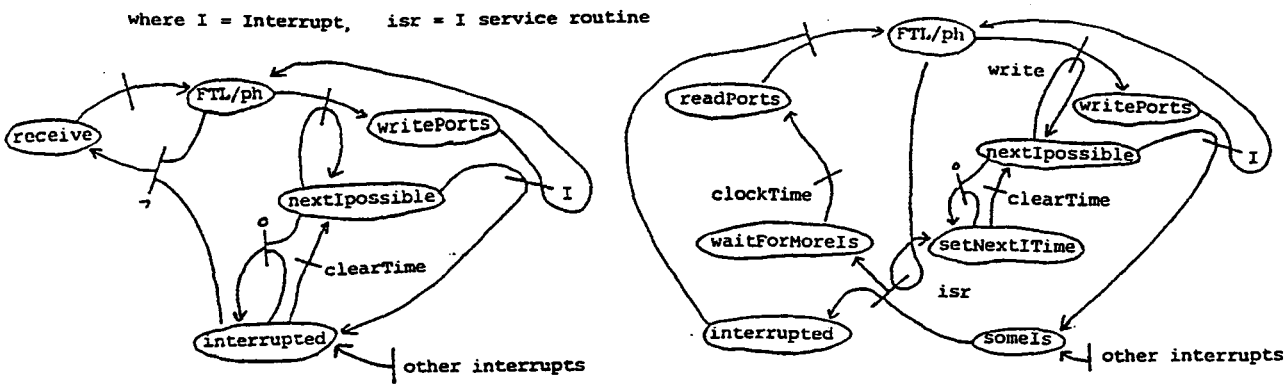


Abb. 3.4.7 Sende- und Empfangsspezifikation (aus [RIS4])

Allerdings mußte dazu eine um den Begriff der *Zeit* erweiterte Modellklasse der Petrinetze verwendet werden, um die Time-Out Bedingungen modellieren zu können. Dies bedeutet aber, daß bestimmte Invariantenbedingungen des einfachen Petriemodells in dieser Modellklasse keine Gültigkeit mehr haben. Trotzdem konnte für das obige Netz für zwei SBCs (Senden und Empfangen) ein Zustandsgraph aus 38 Zuständen erstellt und gezeigt werden, daß die Kommunikation live- und deadlock-frei ist und kein "verhungern" (*starvation*) eines SBC möglich ist [RIS4]. Diese Aussagen lassen sich auch auf jede beliebige Anzahl kommunizierender SBCs erweitern.

Die Verifikation von Programmen ist heutzutage noch ein sehr unerforschtes Gebiet und ist deshalb Gegenstand weltweiter Forschungsbemühungen. Ein Beispiel dafür ist das ProCoS-Projekt (*Provable Correct Systems*), das mit der Beteiligung von 6 Universitäten aus 3 Ländern (ESPRIT Programm, #3104) unter Mitwirkung von C.A.R. Hoare zur Zeit versucht, die Grundlagen für eine systematische Verifikation auf allen Ebenen zu legen.

Allerdings existiert für die Gültigkeit solcher Verifikationssysteme ein Vorbehalt, der nicht übersehen werden sollte und eine gründliche Testphase durchaus nicht überflüssig werden läßt:

Weder die oberste, allgemeine Spezifikation (I/O Spezifikation, s. Abb. 3.4.6) noch die spezielle, dazu konsistenten Spezifikationen der unteren Schichten oder die Formulierung mit Petri-Netzen garantieren, daß die Spezifikation als solche korrekt und vollständig ist. Die Abbildung der gewünschten Systemfunktionen auf die abstrakten Spezifikationsformulierungen durch Menschen unterliegen - wie alle menschlichen Sprachäußerungen - auch den menschlichen Fehlleistungen.

Einen wirksamen Schutz dagegen kann und soll auch die Verifikation nicht leisten. Hier bieten sich andere Möglichkeiten an, Fehler zu vermeiden. Ein Beispiel dafür ist die N-Varianten-Programmierung (s. Abschnitt 1.4.1), die mit der Variante "N-Varianten-Spezifikation" Ergebnisse zeigt, die von den Programmierfähigkeiten der Beteiligten ziemlich unabhängig sind [KEL]. Trotzdem gibt es auch hier Täuschungen, der eine Mehrheit von Programmierern unterliegen [GRAM].

4.0 Parallelität und Fehlertoleranz mit Neuronalen Netzen

Die rasante Entwicklung des Preis-Leistungsverhältnisses der Mikroprozessoren in den letzten Jahren machten Anwendungen möglich, die vorher für Computer undenkbar oder unrentabel erschienen. Dabei zeigte sich aber gleichzeitig, daß bestimmte Anwendungen (Robotik, visuelle Qualitätskontrolle, automatische Sprachübersetzungen, etc.) mit den gängigen von-Neumann Maschinen und den traditionellen Algorithmen sehr schwierig zu bearbeiten waren. Erst der massive Einsatz paralleler Informationsverarbeitung, realisiert durch parallel arbeitende Algorithmen auf parallelen Maschinen, verspricht in naher Zukunft um mehrere Zehnerpotenzen schnellere und vor allen Dingen qualitativ bessere Ergebnisse.

Die Funktionsprinzipien dieser Maschinen beruhen auf der Erkenntnis, daß die traditionellen Rechnerstrukturen Schwierigkeiten haben, den Datenfluß zwischen Prozessor und Speicher in größerem Maße zu steigern (*von-Neumann Flaschenhals*). Es liegt deshalb nahe, soweit wie möglich die Prozessorfunktionen und die Speicherfunktionen zusammenzubringen. Beispiele dafür sind der "rechnende Speicher" von Cherniavsky [CHERN], der flagorientierte ARAM-Assoziativspeicher [WALD] oder die in Abschnitt 2.2 beschriebene "Connection machine". Dabei werden einerseits die Prozessorfunktionen auf wenige, elementare Funktionen reduziert und andererseits die Zahl der unabhängig voneinander arbeitenden Prozessoren stark vergrößert. Verfolgt man diesen Trend weiter, so gelangt man über einfachste, neuronenähnliche Prozessorelemente zu einem der interessantesten Gebiete der Informatik: der Neuroinformatik. Im Unterschied zu den erst seit wenigen Jahren vorhandenen Bemühungen bei den Multiprozessorarchitekturen, gute Modelle für parallele Kontroll- und Datenflüsse zu entwickeln (s. Abschnitt 1.3), beschäftigen die Gehirntheoretiker sich bereits seit ca. 30 Jahren damit, parallele Modelle der Informationsverarbeitung zu entwickeln.

Unser Interesse soll dabei nicht hauptsächlich darin bestehen, neuronal-physiologische Strukturen des Gehirns exakt zu analysieren und nachzubilden und damit sozusagen ein "künstliches Gehirn" zu entwerfen, sondern eher darin, die Grundfunktionen der Informationsverarbeitung von der zellphysiologisch-biologischen Ebene zu abstrahieren und zu modellieren. Sind die überraschenden, teilweise phänomenalen Leistungen des menschlichen Gehirns in ihren Grundlagen geklärt, so lassen sie sich sicher mit den heutigen technischen Mitteln schneller und leichter mit nicht-biologischen Mitteln realisieren. Betrachtet man beispielsweise die Tatsache, daß unsere "Prozessorelemente", die Neuronen, minimale Schaltzeiten von 1ms aufweisen, so kann man mit modernen GaAs Techniken Schaltzeiten im Pikosekundenbereich und mit den in der Entwicklung befindlichen Quanteneffekttransistoren noch einige Zehnerpotenzen schneller schalten. Mit einer dem Gehirn vergleichbaren Integrationsdichte ließe sich dann ein System bauen, das mit einer milliardenfach schnelleren Funktion als das menschliche Gehirn die Bezeichnung "Künstliche Intelligenz" eher verdienen würde als die heutigen wissensbasierten Systeme.

Voraussetzung aber dafür ist, daß man die dafür nötige "Intelligenzstruktur", also die Verbindungsstrukturen der einfachen Prozessorelemente, kennt und sie auch in dieser großen Integrationsdichte verwirklichen kann. Beides sind sehr ehrgeizige Ziele: Die in der Theorie bekannten Modelle und Algorithmen sind noch weit davon entfernt, "intelligentes" Verhalten zu produzieren; ebensowenig wie die heutige VLSI und ULSI Technik die für eine Vernetzung großer Mengen von Einheiten nötigen dreidimensionalen Strukturen bereitstellen kann. Trotzdem gibt es

interessante Ansätze auf beiden Gebieten, die die Hoffnung nähren, daß die gesteckten Ziele prinzipiell nicht unerreichbar sind und deshalb einen engagierten Einsatz lohnen.
Betrachten wir im Folgenden deshalb zunächst die Neuronalen Netze etwas näher.

4.1 Grundlagen Neuronaler Netze

Im Unterschied zur traditionellen Unterteilung der Rechensysteme in die Maschine (Hardware) und in die Algorithmen, die darauf ausgeführt werden (Software), lassen sich bei den Neuronalen Netzen die beiden Aspekte nicht streng voneinander trennen. Ähnlich wie bei den systolischen Feldern sind Hardwarearchitektur und Funktionsalgorithmus als Ganzes zu sehen. Die Hardwarearchitektur implementiert dabei den Algorithmus; die Programmierung als Anpassung des allgemeinen Algorithmus an eine spezielle Aufgabe erfolgt dynamisch durch die Eingabe der Trainingsmuster. Das Gehirn als Vorbild einer neuronalen Maschine wird deshalb manchmal mit einem neuen Namen auch als *Wetware* oder *Brainware* bezeichnet.
Untersuchen wir nun die Funktion der Prozessorelemente, der formalen Neuronen, etwas genauer.

Das Grundmodell

Im Unterschied zur Biologie bzw. Neurologie benutzen wir für unsere Neuronen-Elemente kein Modell, das alle Aspekte eines Neurons exakt beschreibt, sondern nur ein Modell, das eine sehr grobe Verallgemeinerung darstellt. Die sich damit ergebenden Netze sind auch keine Neuronen-Netze, sondern nur "neuronale", also Neuronen-ähnliche Netze. Trotz aller vereinfachenden Annahmen erhofft man sich natürlich trotzdem, noch alle wesentlichen Funktions-Charakteristika übernommen zu haben.

Das Grundmodell eines Neurons stützt sich im Wesentlichen auf das Modell von McCulloch und Pitts [McCUL] aus dem Jahre 1943, das ein Neuron als eine Art Addierer mit Schwellwert betrachtet. Die Verbindungen (*Synapsen*) eines Neurons nehmen eine Aktivierung x_i mit einer bestimmten Stärke w_i auf, summieren diese und lassen dann am Ausgang y (*Axon*) des Neurons eine Aktivität entstehen, sofern die Summe vorher einen Schwellwert t überschritten hat (s. Abb. 4.1.1)

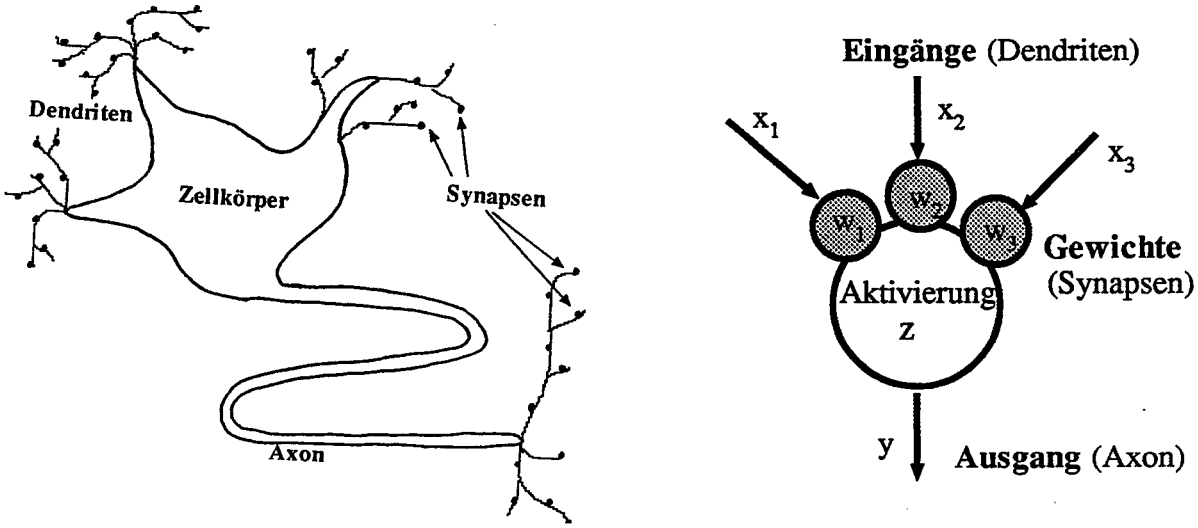


Abb. 4.1.1 Ein biologisches Neuron und ein Modellneuron

Dabei nahmen McCulloch und Pitts noch alle Gewichte als gleich an; eine einzelne Inhibition (negative Gewichte) verhindert die gesamte Ausgabe, die in Übereinstimmung mit den damaligen Erkenntnissen als binär angenommen wurde. Diese Modellierung vernachlässigt natürlich solche Faktoren wie bidirektionale, elektrische Synapsen, chemische Informationswege wie hormonelle Stimulierung bzw. Dämpfung, Energie-Versorgungsfragen wie den Kalzium-Ionenstrom oder die Acethylcolin-Synthese und vieles mehr. Trotzdem ermöglicht dieses einfache Modellierung einige interessante Netzfunktionen.

McCulloch und Pitts zeigten in ihrer Arbeit, daß mit diesen einfachen Elementen jeder finite logische Ausdruck berechnet werden kann. Tatsächlich entsprechen die McCulloch&Pitts Neuronen auch eher den logischen Gattern der Jahrzehnte später erfundenen Computern; mit einer Schwelle von t=1 wird aus einem solchen Neuron ein ODER Gatter, mit t=n bei n Eingaben ein UND Gatter und mit negativen Gewichten ein Inverter.
Nach heutigen Erkenntnissen wird allerdings Information über die absolute Größe des summierten Signals durch die Frequenz der binären Ausgangsimpulse weitergegeben (Frequenz-Modulation); die gewichtete Eingabeaktivität ist dabei in weiten Bereichen der Ausgangsfrequenz proportional.

Modellvarianten

Fassen wir die Eingabeaktivitäten $x_1 .. x_n$ zum Vektor $\mathbf{x} = (x_1, .., x_n)$ und die Gewichte $w_1 .. w_n$ zum Gewichtsvektor $\mathbf{w} = (w_1, .., w_n)$ zusammen, so läßt sich die Summe z der Aktivität im Modellneuron als Skalarprodukt beider Vektoren schreiben:

$$z := \sum x_i w_i = \mathbf{x}\,\mathbf{w}$$

Die Aktivität y am Neuronenausgang wird durch die Transferfunktion T(.), abhängig von der Aktivität z und der Schwelle t beschrieben:

$$y = T(z,t)$$

Der Wertebereich der verwendeten Variablen ist, je nach Modellvariante und Anwendungsbereich, sehr unterschiedlich.

Binäres Modell

Im erweiterten Modell von McCulloch und Pitts sind nur binäre (*aktiv / nicht-aktiv*) Werte (binäre Impulse: *spikes*) für Input x_i und Output y vorgesehen; die Gewichte w_i und die Schwelle t sind dabei reell.

$$x_i, y \text{ aus } \{0,1\} \quad w_i \text{ aus } \Re, \quad y = T_B(z-t) \quad \text{mit } T_B(z) = \begin{cases} 0 & z \leq 0 \\ 1 & z > 0 \end{cases}$$

Anstelle von 0 wird auch für "nicht-aktiv" manchmal der Wert -1 verwendet.

Suprathreshold-linear

Betrachten wir als Aktivität die Impulsfrequenzen, die sich in bestimmten Grenzen durch positive, reelle Zahlen modellieren lassen. Fügen wir nun noch die inhibitorische Aktivität durch negative reelle Zahlen hinzu, so erhalten wir ein Modell, bei dem Input und Output reell sind; die Ausgabe ist proportional zu der Eingabe nach Überschreiten einer Schwelle.

$$x_i, y, w_i \text{ aus } \Re, \quad y = T_L(z,t) := (z-t)\, T_B(z-t)$$

Allgemeine Transferfunktionen

Anstelle einer Stufenfunktion lassen sich auch andere nichtlineare Funktionen (*Squashing functions*) als Schwellwertfunktionen verwenden, die auch die bei großen Signalstärken beobachteten neurologischen Sättigungseffekte modellieren, beispielsweise die differenzierbare, sigmoide Funktion

$$T(z) := (1+\exp(-z))^{-1}$$

Wichtig dabei ist, daß die Funktionswerte sich nicht-konstant von 0 bis 1 erstrecken.

Klassifizierung und Mustererkennung

Formale Neuronen lassen sich sehr effektiv bei dem Problem der Mustererkennung zur Trennung der sogenannten *Musterklassen* verwenden. Betrachten wir dazu als Beispiel Patienten mit chronischer Bauchspeicheldrüsenentzündung, die sich von normalen Patienten durch Konzentrationen bestimmter Stoffe x und y im Blut unterscheiden lassen. In Abbildung 4.1.2 sind die chronischen Patienten (x) und normale Patienten (·) jeweils als Punkte (x,y) repräsentiert.

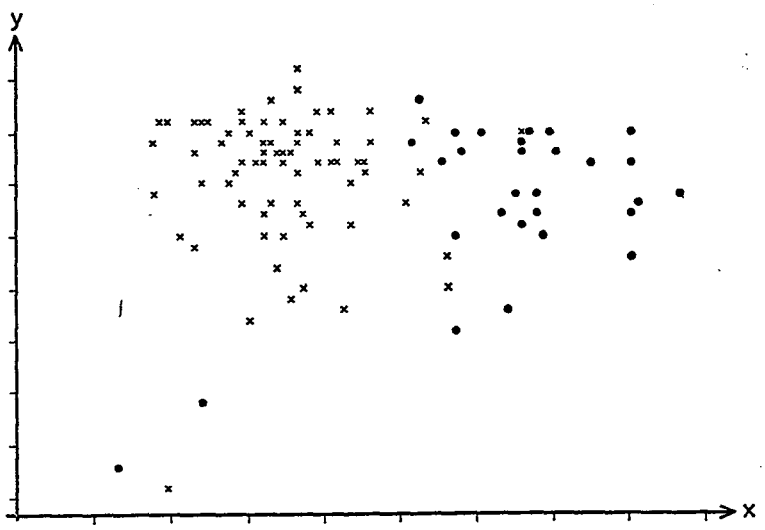


Abb. 4.1.2 Repräsentation von Patienten als Punktmuster

Aufgabe einer Mustererkennung ist es, zwischen den beiden Patientengruppen (*Klassen*) eine Trennungsline (*Klassengrenze*) zu finden, die sie trennt. Für eine Beurteilung eines unbekannten Patienten reicht es dann aus, die Lage seines Musterpunkts diesseits oder jenseits der Klassengrenze zu bestimmen.

Nehmen wir an, es existiere eine einfache Gerade als Klassengrenze (*lineare Separierung*) wie in Abbildung 4.1.3. Dann lautet die Geradengleichung allgemein

$$y = w_0 + w_1 x \qquad x_2 := y, \quad w_2 := -1, \; t := -w_0$$

oder

$$0 = w_0 + w_1 x_1 + w_2 x_2 = \mathbf{wx} - t =: g(\mathbf{x}) \qquad (4.1.1)$$

Es läßt sich also eine *Diskriminierungsfunktion* g(x) definieren, die an der Klassengrenze Null ist. Alle Muster x', die überhalb der Gerade in Klasse 1 liegen, erfüllen die Relation

$$y' > w_0 + w_1 x' \quad \text{oder} \quad 0 > g(\mathbf{x}')$$

und für die Klasse 2 gilt entsprechend $0 < g(\mathbf{x}')$.

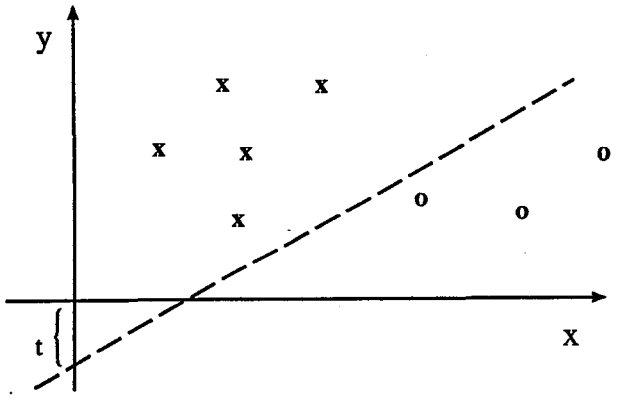


Abb. 4.1.3 Lineare Separierung von Musterklassen

Die Diskriminierungsfunktion wirkt somit wie ein binärer Klassifikator von unbekannten Mustern x.

Vergleichen wir die Form von g(x) mit der Funktion des vorher eingeführten, binären Neuronenmodells, so ist ersichtlich, daß die Funktion y(x) = T(wx-t) eines solchen Neurons gerade eine binäre Diskriminierungsfunktion darstellt. Ein solchermaßen definiertes, formales Neuron stellt also einen Klassifikator dar; jedes Muster wird in eine durch die Gewichte und den Schwellwert definierte Klasse eingeordnet. Durch entsprechende Algorithmen lassen sich die Koeffizienten w, und damit die Klassentrennung, lernen. Ein Beispiel dafür ist der Perzeptron-Algorithmus [MIN].

Betrachten wir nun in Abbildung 4.1.4 eine kompliziertere Klasseneinteilung, die zwei nicht-zusammenhängende Klassen enthält.

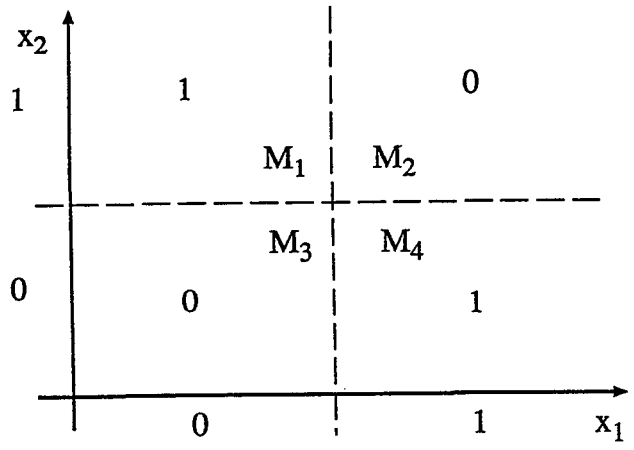


Abb. 4.1.4 Zwei nicht-zusammenhängende Klassen

Diese Situation ist beispielsweise für die 'exklusiv-oder' Funktion XOR (x_1,x_2) gegeben. Wie man leicht sehen kann, lassen sich die beiden Klassen XOR(.)=0 und XOR(.)=1 nicht durch eine Gerade trennen, sie sind *nicht linear separarierbar*. Für dieses Problem gibt es also keine Koeffizienten w und t, die eine solche Klassifizierungsfunktion XOR implementieren würde. Die Beschränkung, die unter anderem von Minsky und Papert in ihrem klassischen Buch *Perzeptrons* [MIN] als Funktionsgrenze der auf den formalen Neuronen beruhenden Systeme aufgezeigt wurde, ließ das Interesse für Jahre an dem Gebiet der neuronalen Netze bis auf wenige Aktivitäten zurückgehen.

Erst neuere Konzepte, wie das der mehrschichtigen Netzwerke (z.B. *Multilayer-Perzeptrons*),

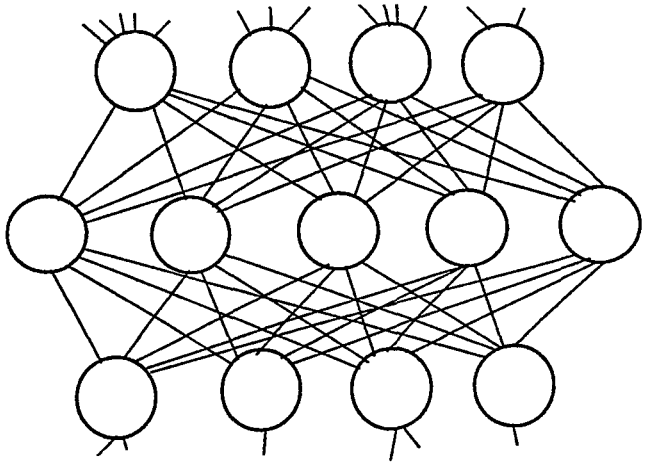
zeigten Wege auf, das obige Problem zu überwinden. Dabei geht man davon aus, daß jede Gerade eine Klassengrenze darstellt, die von einem formalen Neuron implementiert werden kann.
Betrachten wir dazu unser Beispiel der XOR- ähnlichen Musteraufteilung. Zwei parallel arbeitende Neuronen können jeweils die Unterscheidung zwischen den Klassenteilen M_1M_2 / M_3M_4 und M_1M_3 / M_2M_4 realisieren. Aus den Zuständen dieser Neuronen läßt sich nun wiederum mit Hilfe einer UND Verknüpfung durch jeweils ein Neuron auf jeweils ein Klassenstück M_i schließen. Wird nach die Schicht aus vier Neuronen ein weiteres Neuron (eine weitere Schicht) geschaltet, das eine ODER Verknüpfung realisiert, so kann dieses Neuron auf eine beliebige Kombination der M_i - und damit der Klassenform - ansprechen. Dies läßt sich formal folgendermaßen zeigen:
Seien "+", "." und " $\overline{\ }$ " Bool'sches ODER, UND und NICHT, so sind die Ausgaben y, y', y" der drei Schichten

1.Schicht: $y_1 = M_1 + M_2 \quad \overline{y_1} = M_3 + M_4$
$y_2 = M_2 + M_4 \quad \overline{y_2} = M_1 + M_3$

2.Schicht: $y_1' = y_1 \cdot \overline{y_2} = M_1 \quad y_2' = y_1 \cdot y_2 = M_2 \quad y_3' = \overline{y_1} \cdot \overline{y_2} = M_3 \quad y_4' = \overline{y_1} \cdot y_2 = M_4$

3.Schicht: $y'' = \text{Klasse1} = M_1 + M_4 = y_1' + y_4' \qquad y_3' + y_2' = M_2 + M_3 = \text{Klasse2} = \overline{\text{Klasse1}} = \overline{y''}$

Verfügt man direkt über die binären Variablen y_1, y_2, so läßt sich die XOR Funktion nur mit der 2. und 3. Schicht, also in zwei Schichten, darstellen.
Die Konstruktion von Multi-Layer Perceptrons erlaubt also, auch kompliziertere Klassenformen durch stückweise lineare Separierung zu trennen. Die Anzahl der Schichten und die Zahl der Neuronen pro Schicht hängt, wie an dem Beispiel deutlich wird, stark von dem betrachteten Klassifizierungsproblem ab. Wird der Musterraum wie im Beispiel durch Geraden (i.A. Hyperebenen) klassenmäßig zerteilt, so sind in der ersten Schicht soviele Neuronen wie Hyperebenen nötig, in der zweiten Schicht so viele Neuronen wie Klassenteile und in der dritten Schicht so viele Neuronen wie Klassen. Es ergibt sich ein mehrschichtiges Netzwerk, bei dem alle Eingänge einer Einheit einer Schicht mit allen Ausgängen der vorigen verbunden sind, siehe Abbildung 4.1.5.



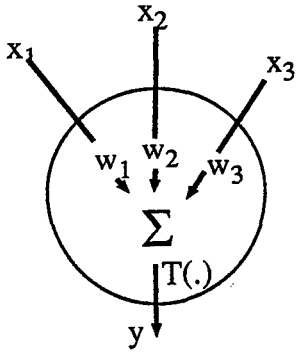


Abb.4.1.5 Mehrschichten Netzwerk Eine Einheit

Dieses Ergebnis läßt sich verallgemeinern. Es läßt sich zeigen [HOR], daß jede beliebige, skalare Funktion f(x) von p Mustern x^1 .. x^p durch ein Netzwerk von zwei Schichten dargestellt werden kann, wobei die erste Schicht aus p Neuronen und die zweite aus einem Neuron besteht.
Darüberhinaus zeigten die Autoren, daß allgemein jede beliebige kontinuierliche Funktione f(x) beliebig dicht durch ein zweischichtiges, rückkopplungsfreies Netzwerk approximiert werden kann. Dabei kann in der 1.Schicht jede beliebige Schwellwertfunktion $T_i(w_i x - t_i)$ verwendet werden, während die 2.Schicht nur die gewichtete Summe $y = \Sigma_i w_i T_i$ ausgibt.
Auch bei einer Verallgemeinerung der Ausgabe von dem Skalar y zu einem Vektor $y = (y_1,..,y_m)$ von Ausgabewerten gelten die obigen Aussagen. Ein interessanter Spezialfall ist eine Cosinus-Schwellwertfunktion: hier stellt das Netzwerk eine diskrete Fouriertransformation dar [GAL].

4.2 Parallelarbeit in Neuronalen Netzen

Im vorigen Abschnitt wurden Modelle für die neuronalen Funktionseinheiten (Prozessoren) entwickelt und ihr Zusammenschalten zu einem System zur Klassifizierung und Mustererkennung vorgestellt. Dabei funktionierten alle Einheiten unabhängig voneinander, so daß sich eine parallele Bearbeitung der Daten mit *parallelem* und *sequentiellen Datenfluß* (s. Abschnitt 1.3.3) ergibt. Interessanterweise gibt es auch Ansätze in der "klassischen", symbolorientierten Künstlichen Intelligenz, die eine Parallelisierung ihrer Regelsysteme auf diesem Weg (unter anderem Namen) erhoffen. Beispielsweise wurde in [PERL] jeder Regel, die bei bestimmten Nebenbedingungen erfüllt ist, ein eigener Prozessor zugeordnet, der bei Regelerfüllung "feuert". Dies entspricht einem formalen Neuron mit festen Gewichten, wie es im vorgen Abschnitt 4.1 eingeführt worden ist.

Ein weiterer, hochinteressanter Aspekt ist die Frage, wie man für die Lösung eines Problems die geeigneten neuronalen Gewichte findet. Im Unterschied zu anderen wissensbasierten Systemen (*klassische Expertensysteme*) bieten die neuronalen Netze die Möglichkeit, durch geeignete Algorithmen die Gewichte anhand von Beispielen "selbstständig" und unüberwacht zu *lernen*. Das Lernsystem (Lernen mit oder ohne Lehrer) und die Form der Gewichtsveränderungen sind dabei entscheidend für die Güte des Lernalgorithmus.

Das Schichtenmodell

Für eine wichtige Netzwerkklasse, den *Feed-forward Netzen*, lassen sich die Einheiten mit gleicher Funktion zu einer einzigen, funktionellen *Schicht* zusammenfassen und mit anderen Schichten pipeline-artig verbinden. Diese Architektur wird vielen Problemen der künstlichen Intelligenz gerecht; Beispiele in der Bild- und Sprachverarbeitung sind in Abb. 4.2.1 gezeigt.
Führt man solches Konzept der Informationsverarbeitung in Schichten oder Funktionsmodulen ein, so stellen sich natürlich Fragen, wie

- *Ist die Funktion der Schicht optimal?*
- *Ist die Funktion effizient parallel implementierbar?*

Da die zweite Frage vom konkreten Algorithmus abhängt, wird sie später näher behandelt werden. Zuvor soll aber kurz auf die Frage nach einer optimalen Funktion einer Schicht eingegangen werden.

Abb. 4.2.1 Schichten der Bild- und Sprachverarbeitung (aus [BR10])

Optimale Schichten

Für das Kriterium einer *optimalen* Funktion lassen sich verschiedene Optimalitätsforderungen aufstellen. Eine der ersten kamen von E. Pfaffelhuber [PFAF], der als Optimalitätskriterium die *Maximierung der 'subjektiven Information'* (gleichbedeutend mit der *Minimisierung der 'subjektiven Entropie'*) vorschlug.

In neuerer Zeit wurde die Idee einer optimalen Informationsverarbeitung von Linsker [LIN1], [LIN2] mit dem Optimalitätskriterium der *maximalen Informationstransmission* einer Schicht, gleichbedeutend mit der *maximalen Varianz* der Ausgabe, wieder aufgenommen. Mit der Überlegung, daß die maximale Varianz auch durch eine Hauptachsentransformation des Input-Musterraums erreicht wird, wobei die Gewichte normiert bleiben müssen, zeigte er, daß die Forderung nach maximaler Information damit gleichbedeutend ist mit einer *Transformation auf die Eigenvektoren* und Eigenwerte des Erwartungswerts der Korrellationsmatrix des Inputs. Damit stellt sich für dieses Optimalitätskriterium die stochastische Approximation der Eigenvektoren und Eigenwerte [OJA1], beispielsweise durch eine iterative Karhunen-Loeve Expansion oder durch eine modifizierte Hebb-Regel [OJA2], als geeigneter Lernalgorithmus heraus (s. z.B. Sanger [SAN]).

Dies ist in guter Übereinstimmung mit anderen theoretischen Ergebnissen von Baldi und Hornik [BALD], die zeigten, daß auch das bekannte Back-Propagation Verfahren (Fehler-Rückführung) zur Minimisierung des quadratischen Fehlers (Varianz!) ebenfalls das Fehlerminimum mit den Eigenvektoren und Eigenwerten der Kovarianzmatrix erreicht.

Ein anderer Vorschlag stammt von M.Bichsel und P.Seitz [BICH], die als Kriterium die *Minimisierung der bedingten Klassenentropie* einführten und damit sogar einen Algorithmus konstruierten, um die Neuronenzahl und die Werte der Gewichte einer Schicht zu optimieren.

4.2.1 Parallelarbeit in topologie-erhaltenden Abbildungen

Die Algorithmen der Neuronalen Netze haben meist nur eine einzige Formulierung mit einer Mischung aus parallelen und sequentiellen Operationen. Da die Simulation dieser Algorithmen bisher fast ausschließlich auf sequentiellen von-Neumann Maschinen durchgeführt worden ist, wurde sehr wenig Augenmerk auf diese Inkonsistenz gerichtet.
Als Beispiel für eine effektive Parallelarbeit sei nun der Algorithmus der topologie-erhaltenden Abbildungen beschrieben, der neben einer gemischten sequentiell-parallelen Form auch in einer rein parallelen Form bekannt ist.

Topologie-erhaltende Abbildungen
Schon früh in der Geschichte der Mustererkennung stellte sich die Notwendigkeit heraus, die Daten, die nur als n-dimensionale Punkte im n-dimensionalen Raum definiert waren, auch für Menschen erfaßbar und begreifbar im 2-dimensionalen Raum mit einer Zeichnung zu illustrieren. Eine Abbildung, die den Musterraum reduziert, verliert dabei sicher Information. Um die Probleme der Trennung der Musterklassen weiterhin zu verdeutlichen, sollte allerdings eine solche Abbildung eine wichtige Eigenschaft besitzen: benachbarte Punkte sollten auch benachbart bleiben. Diese Nachbarschaftserhaltung kann auch mit dem Begriff *Topologie-Erhaltung* beschrieben werden, der die Erhaltung zusammenhängender Punktmengen ausdrückt. Bei den hier betrachteten Algorithmen und Anwendungen wird dieser Begriff allerdings nicht im qualitativen Sinn, sondern eher im quantitativen Sinne eines Gütekriteriums benutzt: Obwohl eine Kugel topologisch nicht äquivalent einem Torus ist [ARN], läßt sie sich trotzdem so auf die Punktmenge eines Torus abbilden, daß das mittlere Fehlerquadrat dabei minimal, aber endlich, wird.
Im folgenden Beispiel aus [BLOM] in Abbildung 4.2.2 ist die Projektion einer 10-dimensionalen Kugel (runde Punkte), umhüllt von einer Oberfläche (Kreuze), auf eine 2-dimensionale Oberfläche dargestellt. In der linken Zeichnung ist dies direkt durch einen einfachen Schnitt entlang der x_3-x_4 Koordinaten, in der rechten Abbildung durch eine nachbarschaftserhaltende Abbildung (*locally sensitive mapping*) geschehen.

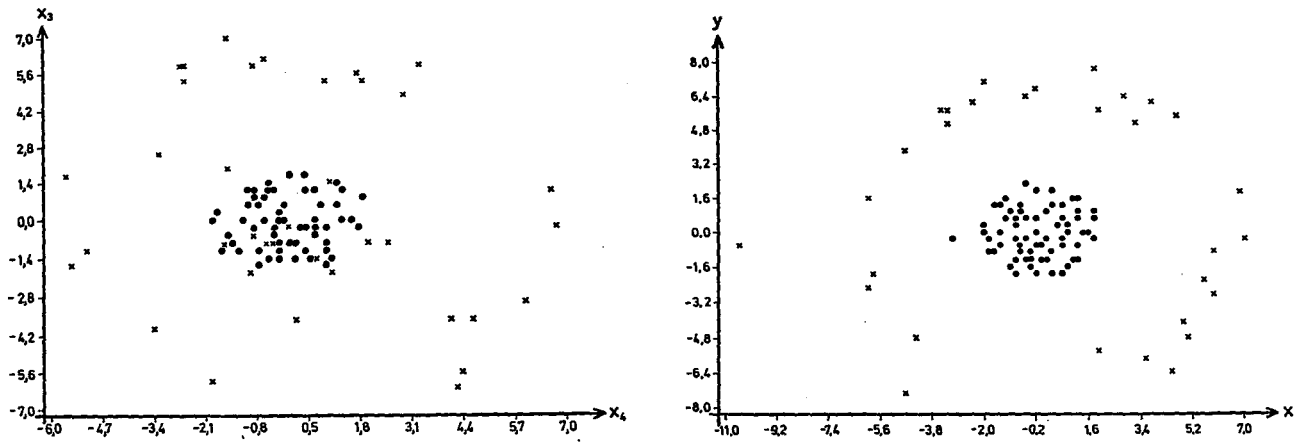


Abb.4.2.2 Einfache Projektion und nachbarschaftserhaltende Abbildung eines 10- dimensionalen Kugelmusters (·) mit Oberfläche(x)

Dabei verwendeten die Autoren allerdings eine Optimierungstechnik, bei der der relative quadratische Fehler zwischen den Punktabständen aller Punkte im alten und im neuen Musterraum minimisiert worden ist. Dies ist eine ziemlich aufwendige Prozedur mit einer hohen Laufzeitkomplexität bezüglich der Zahl der Punkte. Besser ist es, den gesamten Musterraum einer Abbildung zu unterwerfen. Dieses Problem löste Kohonen [KOH3], indem er stellvertretend ein Netz von Einzelpunkten der Optimierung der Nachbarschaftserhaltung unterwarf. Dies läßt sich am Besten in der sequentiellen Version seines Algorithmus verstehen, der im Folgenden beschrieben werden soll.

Ein sequentielles Modell

Seien N Punkte $\mathbf{w}_1 ... \mathbf{w}_N$ gegeben, die beispielsweise in einer zwei-dimensionalen Nachbarschaft zusammenhängen sollen. Assoziieren wir mit jedem Punkt (Gewichtsvektor) eine Prozessoreinheit (Neuron), so hat bei einer einfachen rechteckigen Netzmasche jedes Neuron vier direkte Nachbarn. Jedes Neuron bekommt parallel die gesamte Eingabeinformation (n-dim. Mustervektor). In Abbildung 4.2.3 ist eine solche Konfiguration gezeigt.

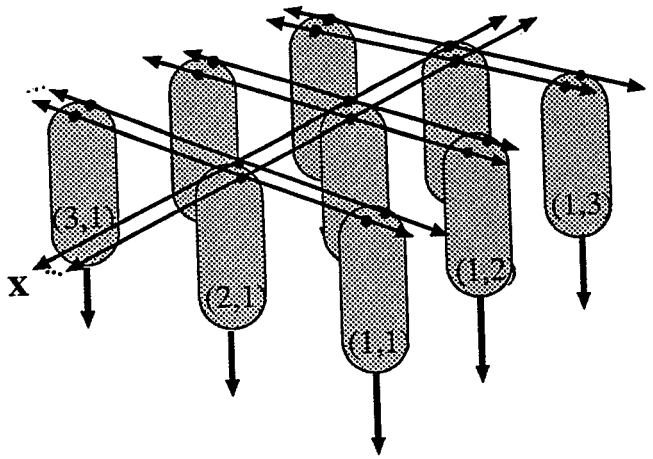


Abb. 4.2.3 Nachbarschaftskonfiguration der Prozessoren $\mathbf{y} = (i_1, i_2)$

Zu Anfang wird nun ein Neuron $\mathbf{y}_c$ ausgewählt, dessen Gewichtsvektor $\mathbf{w}_c$ den kleinsten Abstand zum Eingabemuster hat,

$$|\mathbf{x}-\mathbf{w}_c| = \min_k |\mathbf{x}-\mathbf{w}_k| \qquad \textit{(nearest neighbourhood classification)} \qquad (4.2.1)$$

Die Aktivierung dieses Neurons nach dem Konkurrenzprinzip (*winner-take-all*) bedeutet eine Quantisierung des Eingabemusters (*Vektorquantisierung*): alle geringfügigen Variationen dieses Musters werden auf ein und denselben Gewichtsvektor abgebildet. Mit der Vorschrift (4.2.1) wird somit eine Aufteilung des Musterraums in einzelne *Klassen* vorgenommen; der Gewichtsvektor ist der *Klassenprototyp*.

Die Auswahloperation (4.2.1) wird üblicherweise sequentiell vorgenommen. Eine parallele Hardwareversion ist nur in ziemlich komplizierter Weise denkbar, beispielsweise durch einen Ergebnis-Bus aller Einheiten mit einer Rückkopplung, bei dem alle Einheiten mit geringerem Ergebniswert ihre Ausgabe von selbst abschalten.

Nehmen wir zum Beginn des Algorithmus Zufallswerte für die Gewichtsvektoren an, so geschieht der eigentliche Mechanismus der nichtlinearen Abbildung durch eine Korrektur (Lernen) aller Gewichtsvektoren durch die Eingabemuster **x** zum Zeitschritt t+1:

$$\mathbf{w}_k(t+1) = \mathbf{w}_k(t) + \gamma(t+1)\, h(t+1, c, k)\, [\mathbf{x} - \mathbf{w}_k(t)] \qquad \textit{für alle } k \qquad (4.2.2)$$

Betrachten wir die Gleichung (4.2.2) ohne Term h(t+1,c,k), so findet hier eine *stochastische Approximation* statt, deren Konvergenz unter bestimmten Voraussetzungen für **x** und γ (s. z.B.[LJUN]) gegeben ist. Allerdings ist das Konvergenzziel dabei nicht unbedingt eindeutig bestimmt; in [BR2] sind die Bedingungen für eine Mehrdeutigkeit hergeleitet und an einem einfachen Beispiel demonstriert worden.

Die wichtige Idee des Lernschritts in Gleichung (4.2.2), die ihn von den üblichen Verfahren der stochastischen Approximation abhebt, ist die Einführung einer Nachbarschaft um das ausgewählte Neuron c. Mit Hilfe der Nachbarschaftsfunktion h(t+1,c,k) werden nicht nur das Neuron $\mathbf{y}_c$, sondern parallel dazu alle Neuronen $\mathbf{y}_k$ innerhalb der Nachbarschaft am Lernprozeß beteiligt. Dabei kann die Nachbarschaft diskret definiert sein, beispielsweise alle Neuronen innerhalb eines Radius (Abstand) um das Neuron $\mathbf{y}_c$, oder kontinuierlich, beispielsweise durch die Gaußfunktion $h(t,c,k) := \exp(-(\mathbf{y}_c-\mathbf{y}_k)^2/2\sigma^2(t))$ mit der Standardabweichung (Radius) σ.
Mit der Einführung einer begrenzten Nachbarschaft ist nun die Wahrscheinlichkeitsverteilung der **x** für ein Neuron nicht a priori gegeben, sondern verändert sich während der Iteration. Konvergenzziel und -bedingungen sind hier deshalb schwierig exakt zu bestimmen.

In der folgenden Abbildung ist die Entwicklung und Konvergenz einer solchen topologie-erhaltenden Abbildung am Beispiel eines zwei-dimensionalen Musterraums gezeigt. Die Eingabemuster sind gleichverteilt über den Musterraum, der mit der rechteckigen Umrandung angedeutet ist. In den Musterraum sind die Gewichtsvektoren als Punkte eingezeichnet; Gewichtsvektoren direkt benachbarter Neuronen sind jeweils mit einer Linie verbunden.

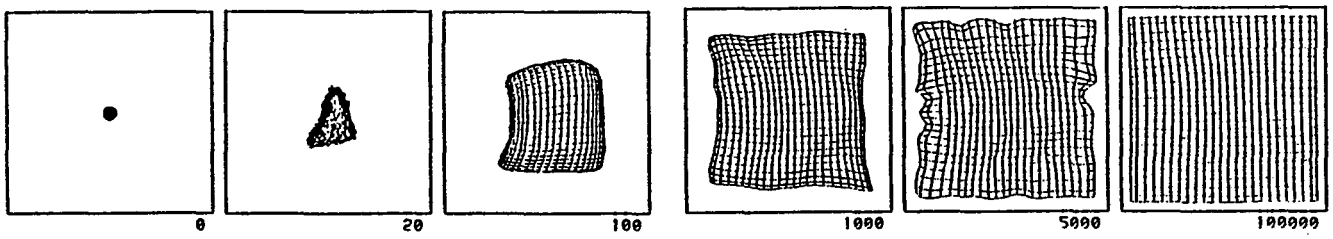


Abb.4.2.4 Selbstorganisation und Konvergenz der topol.-erh. Abbildung (aus [KOH3])

Am Anfang der Iteration haben die Gewichtsvektoren nur Zufallswerte als Initialbedingungen. Schon bald aber wirken sich die über die Nachbarschaft der Neuronen definierten Zusammenhänge aus und die Gewichtsvektoren ordnen sich entsprechend im Musterraum. Die genauen mathematischen Gründe dafür sind sehr schwierig zu erfassen und bisher noch nicht hinreichend analysiert worden. Die Verteilung der Gewichtsvektoren nach der Selbstordnungsphase dagegen ist verschiedentlich näher untersucht worden (s. [KOH2], [RITT2],) und Gegenstand von Kontroversen (vgl. [RITT1]).

Ein paralleles Modell

Das vorher vorgestellte, sequentielle Modell war ursprünglich von Kohonen nur als für eine mathematische Untersuchung besser geeignete Vereinfachung eines parallelen Modells erstellt worden. Das parallele Modell stützt sich im Wesentlichen auf ein physiologisches Phänomen: die *laterale Inhibition.* Hierbei erhalten die Neuronen nicht nur parallel zueinander eine Eingabe, sondern sie hemmen sich untereinander in der weiteren Nachbarschaft, so daß nur wenige, direkt benachbarte Neuronen aktiv übrigbleiben. In Abbildung 4.2.5 ist links ein Querschnitt durch eine solche Neuronenschicht zu sehen; rechts davon ist die fördernde bzw. hemmende Funktion eines Neurons an alle Nachbarn aufgetragen. Wie man sieht, ähnelt die Form der eines Sombreros, so daß sie den Namen *Mexikanerhut*-Funktion erhalten hat. Interessanterweise wird genau diese Funktion auch in der Bildverarbeitung zur Kontrast- und Konturbildung benutzt, s. [LEV],S.180.

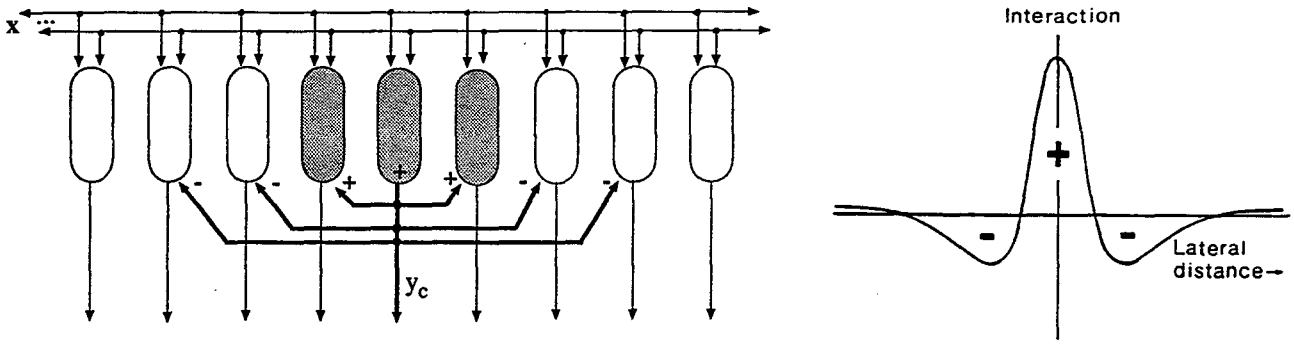


Abb. 4.2.5 Laterale Inhibition und Wechselwirkung mit Nachbarn

Die Neuronen, die dabei aktiv werden, sind nicht mehr vereinzelt, wie dies durch die Vorschrift (4.2.1) gefordert wurde, sondern bilden nun eine ganze Gruppe, bei denen die Gewichtsvektoren ähnlich genug sind. Die Aktivierungsfunktion von Gleichung (4.1.1) ist hier für das k-te Neuron

$$y_k = T(w_k x + \Sigma_i v_{ki} y_i - t_k) \quad \text{mit den Gewichten } v_{ki} \text{ von Neuron k zu i der Mexikanerhut-Funktion} \qquad (4.2.3)$$

Das selektierte Neuron y_c läßt sich nur noch als "Mittelpunkt einer Aktivitätsblase" definieren. In der folgenden Abbildung sind Beispiele davon zu sehen, bei denen die Größe der Aktivität eines Neurons durch die Größe der Punktschwärzung visualisiert wird.

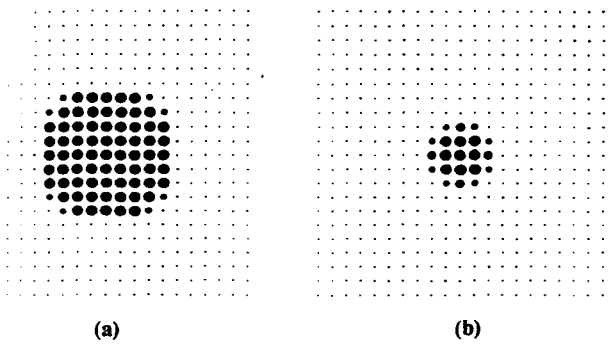


Abb. 4.2.6 Parallele Aktivierung der Neuronen

Optimale Abbildung

Der Algorithmus der topologie-erhaltenden Abbildungen wird von vielen Parametern bestimmt. Was bedeuten die Optimalitätsforderungen des vorigen Abschnitt für die Parameter, beispielsweise die Forderung nach maximaler Informationstransmission ?
Mit dem Eingabemuster $\mathbf{x}$ können wir den Shannon'schen Informationsgewinn I_{trans} bei N Ausgabepunkten $\mathbf{w}_i$ berechnen:

$$I_{trans} = I_{out} - I_{out/inp} = -\ln[P(\mathbf{w}_i)] + \ln[P(\mathbf{w}_i/\mathbf{x})]$$

Mit der Operation < > zur Bildung des Erwartungswerts einer Funktion $\langle f(\mathbf{w}_i)\rangle := \Sigma_{w_i} P(\mathbf{w}_i)\, f(\mathbf{w}_i)$ erhalten wir für die mittlere Informationstransmission

$$\langle I_{trans}\rangle_{w_i,x} = \langle I_{out}\rangle_{w_i,x} - \langle I_{out/inp}\rangle_{w_i,x} = -\Sigma_i P(\mathbf{w}_i)\ln[P(\mathbf{w}_i)] \; - \; \Sigma_x P(\mathbf{x}) \Sigma_i P(\mathbf{w}_i/\mathbf{x})\ln[P(\mathbf{w}_i/\mathbf{x})]$$

Dies wird ein Maximum, wenn

$$\langle I_{out}\rangle_{w_i,x} \stackrel{!}{=} \max \qquad (4.2.4)$$

$$\langle I_{out/inp}\rangle_{w_i,x} \stackrel{!}{=} \min \qquad (4.2.5)$$

Eine hinreichende Bedingung für (4.2.4) wird in [BR15] hergeleitet:

$$P(\mathbf{w}_i) = P(\mathbf{w}_j) = 1/N \qquad \text{für alle } i,j \qquad (4.2.6)$$

Für die Forderung (4.2.5) wissen wir, daß die Werte für $P(\mathbf{w}_i/\mathbf{x})$ sehr ungleich sein müssen, um ein Gesamtminimum zu erreichen. Dies wird am Besten, wie in [BR15] gezeigt, durch eine vollständige Zerlegung des Musterraums in disjunkte Klassen erreicht, wobei nicht gesagt wird, auf welche Weise ein entsprechender Algorithmus dies erreichen soll.
Aus der hinreichenden Bedingung (4.2.6) der gleichen Auftrittswahrscheinlichkeit folgt nun direkt (s. [BR15]), daß als wichtige Bedingung die Klassendichte $M(\mathbf{x})$ (Dichte der Klassenprototypen) direkt proportional zur Wahrscheinlichkeitsdichte der Inputmuster sein muß: $M(\mathbf{x}) \sim p(\mathbf{x})$.
Die bessere Auflösung des Musterraums durch mehr Klassen in Zonen dichterer "Punktwolken" kann man zwar als plausibel empfinden, steht aber im direkten Widerspruch zu den Ergebnissen von Linsker [LIN3], der genau das Gegenteil fand.

Hat nun der topologie-erhaltende Algorithmus von Kohonen diese Optimalitätseigenschaften?

Kohonen fand bei einer Untersuchung [KOH2] allgemein die Proportion $M(\mathbf{x}) \sim p(\mathbf{x})$. Demgegenüber leiteten Ritter und Schulten in [RITT1] mathematisch her und zeigten durch Simulation, daß dies nicht allgemein der Fall ist. Beispielsweise ergeben sich für den eindimensionalen Fall $M(\mathbf{x}) \sim p(\mathbf{x})^{2/3}$ und für höhere Dimensionen kompliziertere Ausdrücke. Nur im zweidimensionalen (komplexen) Fall gilt obige Relation $M(\mathbf{x}) \sim p(\mathbf{x})$, so daß der Algorithmus immerhin im zweidimensionalen Fall *optimal* genannt werden kann.

4.2.2 Anwendungen in der Robotersteuerung

Die Methode der topologie-erhaltenden Abbildungen läßt sich auf die verschiedensten Probleme anwenden. Sie ist nicht beschränkt auf die ursprünglichen Anwendungen in der Mustererkennung (s. vorigen Abschnitt 4.2.1), sondern kann beispielsweise auch zur approximativen Lösung des Problems des Handelsreisenden eingesetzt werden, der mit einer möglichst kurzen Rundreise alle Städte einer vorgegeben Menge besuchen will. In [ANG] ist dieses Problem dadurch gelöst, daß man für jede Stadt ein formales Neuron in eine lineare, geschlossene Kette hängt und die Menge aller Stadtkoordinaten als Inputvektoren auffaßt. Der zwei-dimensionale Inputraum wird damit auf eine ein-dimensionale Kette (Reiseroute) abgebildet, wobei die Nachbarschaftstopologie (benachbarte Städte sind auch in der Kette benachbart) so gut wie möglich (mit dem kleinsten quadratischen Fehler) erhalten bleibt.

Eine andere Anwendung der topologie-erhaltenden Abbildungen ist in der Steuerung von Robotermanipulatoren zu finden. Interessanterweise treten auch bei der menschlichen Muskelsteuerung topologie-erhaltende Abbildungen auf. Die folgende Zeichnung zeigt die Zuordnung der Muskeln zu den Gehirnarealen (*Somatotopische Abbildung*) und daneben einen Roboterarm vom PUMA Typ, wie er bei den weiteren Betrachtungen Verwendung finden wird.

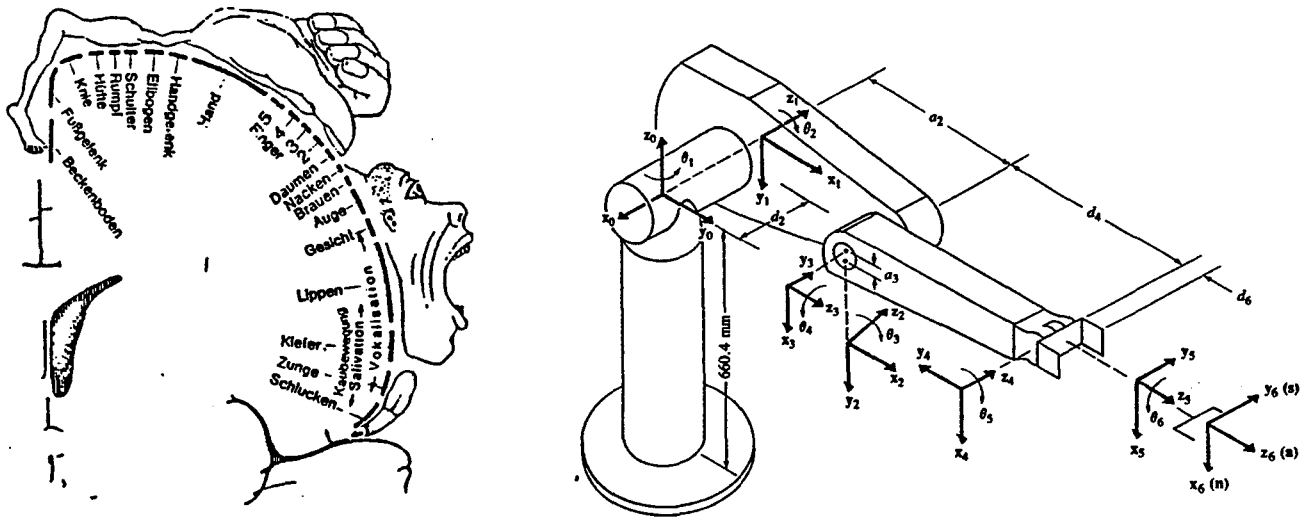


Abb. 4.2.7 Topologie-erhaltende Abbildungen beim Menschen (aus [SCH]) Der PUMA Roboter

Die verschiedenen Kontroll- und Sensorschichten der Modelle der menschlichen Muskelkontrolle lassen sich den Schichten der Roboterkontrolle gegenüberstellen. In Abbildung 4.2.8 ist dies für eine relativ grobe Modellierung gezeigt.

Für die Roboterkontrollschicht der direkten Positionskontrolle läßt sich nun ein direkter Ansatz angeben, um mit Hilfe der topologie-erhaltenden Abbildungen eine Kontrolle der Gelenkwinkel der Manipulatorgelenke im kartesischen Raum zu erlernen, anstatt sie durch analytisch-geometrische Transformationen zu errechnen.

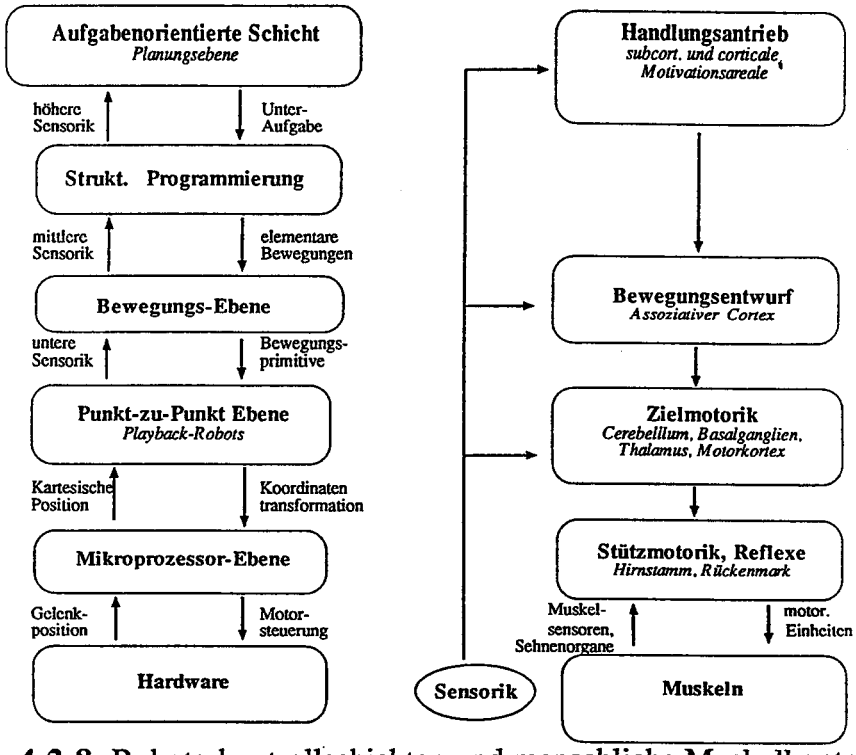


Abb. 4.2.8 Roboterkontrollschichten und menschliche Muskelkontrolle (aus [BR10])

Das Grundproblem der statischen Positionskontrolle oder *Kinematik* besteht darin, bei vorgegeben Gelenkwinkeln θ_i die Arm- bzw. Handwurzelposition **x** zu errechnen. Ordnen wir jedem Manipulatorsegment eine sog. *homogene* Transformationsmatrix [DEN] zu, mit der Translationen und Rotationen beschrieben werden, so können wir die tatsächliche kartesische Position des Arms leicht durch sukzessives Ausmultiplizieren der Transformationsmatrizen erhalten. Jede Transformationsmatrix reflektiert dabei die Geometrie des Segments, zu der sie gehört.

Das umgekehrte Problem, aus der vorgegeben kartesischen Position **x** die Gelenkwinkel θ_i zu bestimmen (*inverse Kinematik*), ist dagegen wesentlich schwerer analytisch zu lösen. Jede Armgeometrie benötigt eine besondere Lösung, die auch nur mit akzeptablen Aufwand zu erreichen ist, wenn die Armgeometrie bestimmten Restriktionen (parallele oder orthogonale Gelenkachsen) erfüllt. Allgemein findet man folgende Probleme (aus [BR15]):

- Δ Alle Änderungen der Armgeometrie (Fabrikänderungen, Abnutzung, etc) ziehen schwierige analytische Änderungen nach sich, die ein normaler Anwender nicht leisten kann. Damit ist eine anwenderorientierte Anpassung des Roboterarms praktisch unmöglich.

- Δ Hat der Manipulator mehr Freiheitsgrade (Gelenke) als der kartesische Raum, so müssen willkürliche Bewegungsrestriktionen ("überzählige" Koordinaten bleiben konstant) eingeführt werden; der Arm bewegt sich nicht mehr optimal.

- Δ Die analytische, inverse Kinematik ist ziemlich komplex und benutzt nur langsam zu berechnende, transzendente Funktionen. Damit wird eine Umrechnung zur Laufzeit sehr problematisch.

- Δ Die Genauigkeit der Positionsauflösung paßt sich nicht der Problemlage an, sondern ist von der Armgeometrie im Arbeitsbereich abhängig.

Δ Die Vermeidung von Hindernissen und Beachtung anderer Restriktionen (Nebenbedingungen) bei der Bewegung ist nicht automatisch, sondern muß extra von Hand programmiert werden.

Diese Probleme lassen sich mit den topologie-erhaltenden Abbildungen vermeiden. Dazu unterteilen wir die Armbewegung in zwei Teile: eine *grobe* Armbewegung, die im Wesentlichen die nicht-lineare inverse Kinematik beinhaltet, und in eine *feine* Armbewegung, die mit Hilfe der gewünschten Kartesischen Position (Eingabemuster) zwischen den groben Positionsrastern interpoliert.

Die grobe Positionierung

Betrachten wir eine topologie-erhaltende Abbildung, wie sie in Abschnitt 4.2.1 eingeführt wurde. Der gesamte Arbeitsraum ist in Zellen (Klassen) unterteilt. Jeder Zelle wird ein Neuron mit seinem Gewichtsvektor zugeordnet, wobei der Gewichtsvektor den Mittelpunktskoordinaten der Zelle entspricht. Jedes Neuron **y** der drei-dimensionalen Nachbarschaft ist durch sein Tripel (i_1,i_2,i_3) von Indizes innerhalb des Netzes gekennzeichnet. In Abbildung 4.2.9 ist dies illustriert.

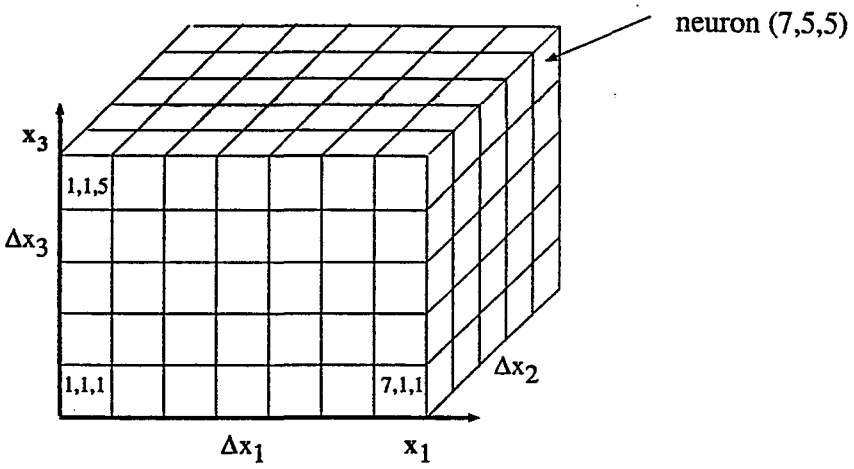


Abb. 4.2.9 Aufteilung des Arbeitsraums

Mit jedem Neuron **y** mit dem Gewichtsvektor **w** des kartesischen Raums sei auch ein entsprechender Vektor $\Theta = (\theta_1,\theta_2,\theta_3)$ des Gelenkkoordinatenraums assoziiert. Wird nun eine Kartesische Position **x** gewünscht, so wird beim sequentiellen Algorithmus mit Gleichung (4.2.1) zuerst festgestellt, welches Neuron $\mathbf{y}_c$ die Abbildung der inversen Kinematik vollbringt. Damit läßt sich sofort auf die dem Gewichtsvektor entsprechende Position Θ in Gelenkkoordinaten schließen, so daß die Transformation der inversen Kinematik hier nur aus einer einfachen, schnellen Tabellen-Ausleseoperation besteht.
Dieses schnelle *memory-mapping* bringt allerdings auch Probleme mit sich:

- Die Tabellenzuordnung hängt stark von der Geometrie des Arms ab und sollte deshalb - um eine analytische Problembehandlung zu vermeiden- erlernbar sein.

- Für jede Zelle des Arbeitsraumes ist nur eine Gelenkposition vorgesehen. Dies führt prinzipiell zu Positionierungsfehlern.

Beispiel: Für einen würfelförmigen Arbeitsraum mit der Kantenlänge von 70 cm und N=1000 Neuronen ensteht ein maximaler Fehler von $7x3^{1/2}$=12.12 cm, was entschieden zu groß für eine normale Roboteranwendung ist.

- Für eine ausreichende Genauigkeit sollten möglichst viele Funktionswerte vorhanden sein. Dies führt aber zu großem Speicherbedarf.

  Beispiel: Bei obigem Arbeitsraum und $N=10^{12}$ Neuronen ensteht ein Fehler von 0.121mm, was noch zu tolerieren ist. Dies bedeutet aber bei 16 Bit pro Zahl ein Speicherbedarf von $2 \cdot 6 \cdot 10^{12}$ Byte = 12 Gigabyte!

Aus den letztgenannten Gründen führen wir eine lineare Korrektur der groben Bewegung ein.

Die Feinpositionierung

Wie schon Ritter und Schulten in [RITT3] vorschlugen, läßt sich der durch die festen Tabellen verbleibende Fehler durch eine lineare Approximation für die Gelenkposition $\Theta(\mathbf{x})$ verkleinern:

$$\Theta(\mathbf{x}) = \Theta_c + \Delta\Theta = \Theta_c + \mathbf{A}_c (\mathbf{x}-\mathbf{w}_c) \qquad (4.2.7)$$

Dabei ist $\mathbf{A}_c$ eine Matrix, deren Koeffizienten für jedes Neuron extra gelernt werden muß. Zwar bedeutet das auch zusätzlichen Speicherbedarf pro Neuron, es verringert aber über die geringere notwendige Zahl von Neuronen trotzdem den Gesamtspeicherbedarf.

Der Lernalgorithmus

Das Lernen der gespeicherten Tabellenwerte für $\mathbf{w}_k$ geschieht mittels stochastischer Approximation nach Gleichung (4.2.2).

Die übrigen Werte Θ_k und $\mathbf{A}_k$, also die 3 Gelenkkoordinaten von Θ_k und die 9 Matrixkoeffizienten A_{11}, ... ,A_{33}, fassen wir zu dem allgemeinen Gelenkparametervektor $\mathbf{u}_k = (\theta_1,\theta_2,\theta_3,A_{11}, \ldots ,A_{33})^T_k$ zusammen. Dieser läßt sich analog zu (4.2.2) folgendermaßen iterieren:

$$\mathbf{u}_c(t+1) = \mathbf{u}_c(t) + h(.)\gamma(t+1)[\mathbf{u}_c^*(t+1)-\mathbf{u}_c(t)] \qquad (4.2.8)$$

mit der Nachbarschaftsfunktion h(.) und der (t+1)-ten Schätzung $\mathbf{u}_c^*$ von $\mathbf{u}_c$.

Was sind gute Schätzungen von Θ_c^* und $\mathbf{A}_c^*$?

Eine Schätzung von Θ_c läßt sich beispielsweise durch den gemessenen Fehler $(\mathbf{x}-\mathbf{x}_F)$ zwischen gewünschter Position $\mathbf{x}$ und tatsächlicher Position $\mathbf{x}_F$ erreichen.

$$\Theta_c^* = \Theta_c + \mathbf{A}_c(\mathbf{x}-\mathbf{x}_F) \qquad (4.2.9)$$

Eine Schätzung für $\mathbf{A}_c$ wurde von Ritter und Schulten [RITT3] vorgeschlagen, wobei außer der tatsächlichen Endposition $\mathbf{x}_F$ auch die tatsächliche Position $\mathbf{x}_I$ nach der groben Bewegung verwendet wird:

$$\mathbf{A}_c^* = \mathbf{A}_c + \mathbf{A}_c((\mathbf{x}-\mathbf{x}_F) - (\mathbf{w}_c-\mathbf{x}_I)) (\mathbf{x}_F-\mathbf{x}_I)^T / |(\mathbf{x}_F-\mathbf{x}_I)|^2$$

Ein anderer, einfacherer Ansatz für $\mathbf{A}_c$, der ohne $\mathbf{x}_I$ auskommt, ist in [BR14] enthalten.

Der prinzipielle Fehler in der Positionierung

Betrachten wir ein System, in dem alles Lernen abgeschlossen ist und das vollständig konvergiert hat. Selbst wenn wir annehmen, daß die stochastischen Abweichungen vom Konvergenzziel vernachlässigbar klein geworden sind, besteht durch die lineare Approximation einer nicht-linearen Funktion prinzipiell ein Positionierungsfehler e^{LA}.

Wie groß ist der Positionierungsfehler ?

Dazu wählen wir uns zweckmäßigerweise eine bekannte Roboterarchitektur aus, beispielsweise die PUMA Konstruktion aus Abbildung 4.2.7, und berechnen den maximalen Positionierungsfehler aus der Differenz zwischen der tatsächlichen Position, errechnet mittels der für das Beispiel bekannten analytischen Lösung, und der approximierten Position, ermittelt durch analytisch bestimmtes Θ_c und den als Differenzenquotienten angenäherten Koeffizienten von **A**.

Dieser Positionierungsfehler wurde in [BR14] entlang eines linearen Weges durch den Arbeitsraum bestimmt und für verschiedene Anzahl von Neuronen n pro Dimension (Neuronen-Gesamtzahl $N=n^3$) als Parameter aufgetragen. In Abbildung 4.2.9 ist links der absolute Kartesische Fehler aufgetragen, rechts die Abhängigkeit des Fehlers von der Anzahl n der Neuronen für einen Punkt des Weges. Wie man sieht, ist die Abhängigkeit gut mit der linearen Approximation $\lg(e^{LA})\sim-\lg(n)$ zu fassen, so daß sich der Fehler e^{LA} der linearen Approximation zu $e^{LA} = C\ n^b$ mit $C:=10^a$, b=-2.6 und C=2.0 ergibt.

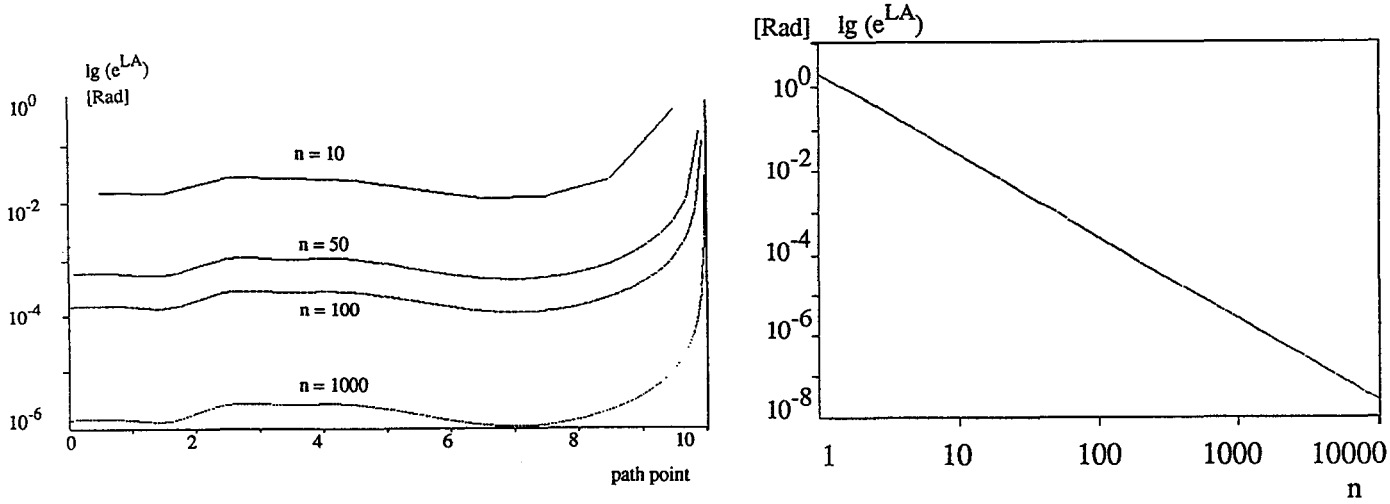


Abb.4.2.9 Absoluter Positionierungsfehler entlang eines linearen Weges und an einem Punkt

Eine weitere Fehlerquelle bildet die endliche Auflösung *r* der gelernten Tabellenwerte für $\mathbf{w}_k$, Θ_k und $\mathbf{A}_k$, die identisch ist mit der *Information* r:= ld(Zahl der möglichen Werte). Der gesamte maximale Positionierungsfehler e^{MAX} ist damit eine Überlagerung aus dem Fehler der linearen Approximation e^{LA} und dem Fehler bei endlicher Auflösung e^{RES}

$$e^{MAX} = | e^{LA} + e^{RES} | \qquad (4.2.10)$$

Die prinzipielle praktische Verwendbarkeit des speicherplatz-intensiven Algorithmus ist

zweifelsohne stark von seinem Speicherplatzbedarf bei noch akzeptablen Positionierungsfehler abhängig. Wie lassen sich nun die verfügbaren Parameter des Systems so einrichten, daß bei gegebenem maximalen Fehler die minimal nötige Informationsmenge (der minimale Speicherplatzbedarf) resultiert?

Optimale Informationsverteilung

Wählen wir bei gegebener, fester Gesamtinformationsmenge eine große Anzahl von Neuronen, so wird der Auflösungsfehler durch die geringen Auflösungen r_w,r_θ und r_A der Tabellenwerte für $\mathbf{w}_c$, $\Theta_c$and $\mathbf{A}_c$ besonders groß. Vergrößern wir dagegen die Auflösungen der Tabellenwerte, so vergrößert sich auch der Fehler der linearen Approximation durch eine resultierende, geringere Zahl von Neuronen (Zahl der Tabellenwerte). Für eine optimale Informationsverteilung definieren wir folgendes Optimalitätskriterium:

> Bei einer *optimalen Informationsverteilung* führt eine kleine (*virtuelle*) Umverteilung von Information (Verändern von r_w,r_θ ,r_A oder n) weder zu einer Erhöhung, noch zu einer Erniedrigung des Positionierungsfehlers.

Eine Konfiguration, bei der eine Veränderung in einer Richtung eine Verkleinerung des Fehlers bewirken würde, ist zweifelsohne nicht optimal, da dann ja auch mit weniger Information der gleiche Fehler erreicht werden könnte. Aber auch eine Erhöhung des Positionierungsfehlers beim Informationstransfer ist nicht günstig: Ein Wechsel der Richtung des Transfers würde dann ebenfalls eine Fehlerabnahme wie im ersten Fall bewirken.

Dieses Optimalitätsprinzip läßt sich nun approximativ mit den ersten Ableitungen formulieren. Es gilt für eine Informationsverschiebung (Änderung des Speichers s) Δs

$$\Delta e^{MAX}(s) = [\frac{\partial}{\partial n} e^{MAX}(n)\frac{\partial}{\partial s} n(s) + \frac{\partial}{\partial r_w} e^{MAX}(r_w)\frac{\partial}{\partial s} r_w(s) + \frac{\partial}{\partial r_\theta} e^{MAX}(r_\theta)\frac{\partial}{\partial s} r_\theta(s) + \frac{\partial}{\partial r_A} e^{MAX}(r_A)\frac{\partial}{\partial s} r_A(s)]\ \Delta s \qquad (4.2.11)$$

Bei einer optimalen Informationsverteilung müssen alle Terme der Summe gleich sein.

Setzen wir für $e^{MAX}(n,r_w,r_\theta,r_A)$ Gleichung (4.2.10) ein und rechnen dies aus, so ergibt sich ein System von drei Gleichungen mit vier Variablen n, r_w, r_θ und r_A. Dies ist in [BR14] gelöst. Berücksichtigen wir noch zusätzlich den Ausdruck für den Speicherplatzbedarf $s = n^3(3r_w+3r_\theta+9r_A)$, so erhalten wir die optimalen Systemparameter für einen minimalen Fehler bei gegebener Speichergröße. Dies ist in Abbildung 4.2.11 links gezeigt. Auf der rechten Seite ist der absolute Wert des Kartesischen Fehlers als Funktion der Speichergröße bei optimaler Informationsverteilung eingezeichnet. Zum Vergleich ist neben der Parameterkonfiguration mit optimalen Auflösungen r_w, r_θ und $r_A$auch die optimale Konfiguration bei konstanter Auflösung $r_w = r_\theta = r_A := r$ eingezeichnet.

Bei einem geringen Kartesischen Fehler und dem hohen Speicherbedarf von $1.13\ 10^{14}$ Bytes ist der Fehler bei konstanter Auflösung 5.6 mal größer als bei variabler Auflösung. Betrachten wir aber eine realistische Konfiguration mit einem Fehler von 0.201 mm, ein Wert, der in der Größenordnung der mechanischen Ungenauigkeit des PUMA Arms liegt, so benötigen die dafür notwendigen 39.6^3 Neuronen mit konstanter Auflösung von r = 16.4 Bit einen Speicherplatz von 1.9MB. Damit hat diese Konfiguration einen nur 18% größeren Fehler als die optimale Konfiguration (0.164 mm) mit den variablen Auflösungen r_w=17, r_θ=20.1, r_A=15 Bits. Bedenkt man den Zusatzaufwand an Software, um Multiplikationen mit verschiedener Auflösung zu unterstützen, so scheint für normale Anwendungen die Benutzung variabler Auflösungen nicht gerechtfertigt.

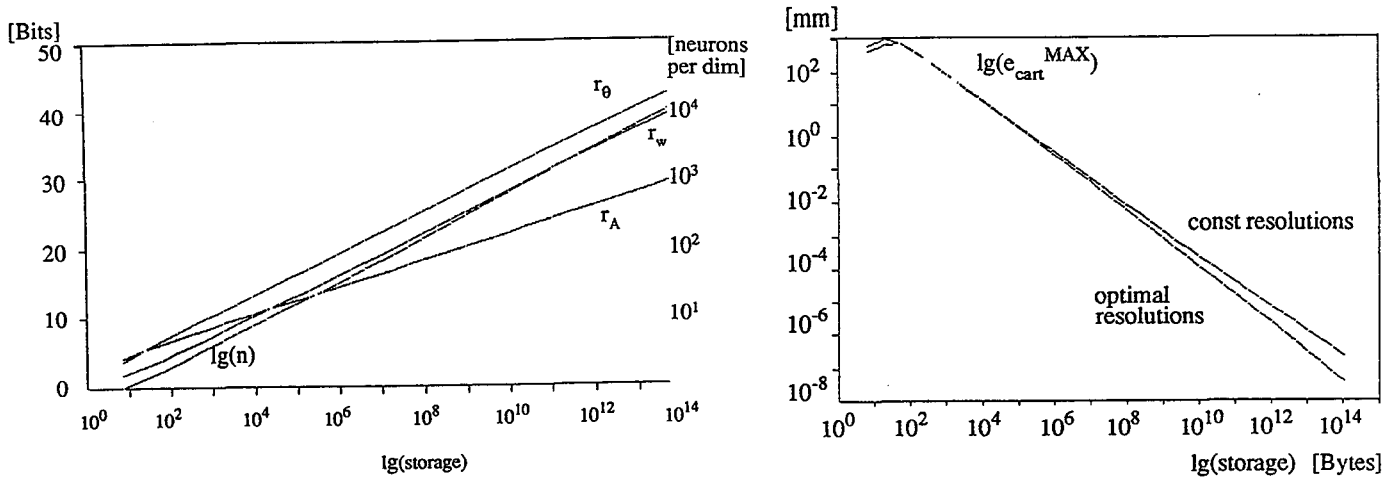


Abb.4.2.11 Optimale Informationsverteilung und maximaler Kartesischer Fehler

Zusammenfassung und Ausblick

In den vorigen Abschnitten wurde gezeigt, wie sich topologie-erhaltende Abbildungen zur Robotersteuerung einsetzen lassen. Die Proportionen von optimalen Abbildungen im Sinne einer maximalen Informationserhaltung wurde diskutiert und ein Algorithmus angegeben, der bei zweidimensionalen Abbildungen optimal ist.

Weiterhin wurde gezeigt, wie sich aus dem Prinzip einer optimalen Informationsverteilung für einen minimalen Positionierungsfehler die optimalen Werte für Neuronenzahl, Positionskodierung und Speichergröße ergeben.

Die analytisch fundierte Roboterpositionierung wird bei dem vorgestellten Verfahren durch das Lernen einer Positionierungstabelle ersetzt. Dies hat folgende Vorteile:

- ♣ Wie bei allen tabellierten Funktionen wird die Kontrolle sehr schnell, da das Ausrechnen der Funktionen durch ein Nachschlagen in Tabellen (memory mapping) ersetzt wird.

- ♣ Da keine analytischen Lösungen für die inverse Kinematik benötigt werden, lassen sich mit dieser Methode auch sehr unkonventionelle Architekturen steuern, bei denen die Gelenkachsen nicht orthogonal oder parallel zueinander orientiert sind oder die mehr Gelenke (Freiheitsgrade) als Kartesische Koordinaten besitzen.

- ♣ Dies bedeutet auch, daß Veränderungen der Armgeometrie für den Einsatz beim Benutzer keine umständlichen, zeitraubenden und kostspieligen Rückfragen beim Hersteller nötig machen; der Benutzer kann ohne tiefere Roboterkenntnisse den Manipulator direkt seinen Bedürfnissen anpassen.

- ♣ Die Auflösung der topologie-erhaltenden Abbildung hängt ortsabhängig von der Zahl der Trainingspositionierungen (Wahrscheinlichkeitsdichte) ab. Der Lernalgorithmus ermöglicht damit automatisch eine Anpassung der Positionierungsauflösung an den eigentlichen Einsatzort.

- ♣ Es lassen sich unproblematisch Nebenbedingungen (minimale Energie, minimale Abweichung vom mittleren Winkel) einführen, indem die Lerngleichungen (4.2.8) entsprechend angepaßt werden.

Dabei sind allerdings auch einige Probleme zu beachten:

- Ein charakteristisches Problem von tabellierten Funktionen ist der Speicherbedarf, um eine gute Auflösung zu erreichen. Auch diese speziellen, topologie-erhaltenden Abbildungen bilden dabei, wie gezeigt wurde, keine Ausnahme. Trotzdem scheint es durch einen speicheroptimierten Ansatz möglich, bei realistischen Problemen wie einer Positionssteuerung eines PUMA Roboters mit relativ moderaten Speichergrößen von ca. 2MB auszukommen.

- Ein wichtiges Problem ist die Zeit, um den Algorithmus durchzuführen. Nehmen wir beispielsweise die obige Konfiguration von ca. 2 MB. Dann sind ca. 40 Neuronen in jeder Dimension, also insgesamt 64000 Neuronen, nötig. Benutzen wir den sequentiellen Algorithmus, so muß bei jeder Positionierung das Abstandsminimum von allen 64000 Gewichtsvektoren gesucht werden, was bei einer Nicht-Gleichverteilung einen ziemlichen Rechenaufwand bedeutet. Nimmt man dagegen eine Gleichverteilung und ersetzt die Minimumsuche bei den resultierenden, gleichartigen Zellen durch das direkte Ausrechnen des Neuronenindexes, so muß man auf eine dynamische, problemangepaßte Resolution verzichten.
  Ein Ausweg aus diesem Dilemma bietet der parallele Algorithmus, der eine parallele Operation von vielen einfachen, in Hardware implementierten Prozessoren erlaubt.

Die erlernte Positionierung mittels topologie-erhaltender Abbildungen läßt allerdings auch einige Probleme außer acht, die Gegenstand weiterer Forschungsanstrengungen sein müssen:

- ∇ Die Positionierung des Manipulators mittels topologie-erhaltender Abbildungen stellt nur einen Ansatz für eine Roboterkontrolle auf *unterster Ebene* dar (s. Abb. 4.2.8). Es ist sehr unklar, wie dieser Ansatz auf höhere Kontrollebenen wie Trajektorienkontrolle u.ä. ausgedehnt werden kann.

- ∇ Die erlernten Transformationstabellen sind *fest* und müssen bei jeder kleinsten Änderung der Manipulatorgeometrie, z.B. einer leichten Schiefstellung, neu erlernt werden. Es gibt keine Bewegungsprimitive, die bei einer solchen Umstellung erhalten bleiben und als Wissen benutzt werden können.

- ∇ *Zeitsequenzen* können nicht erlernt werden; sie obliegen den höheren Kontrollschichten. Damit gibt es auf dieser Ebene auch keine Möglichkeit, beispielsweise dieselbe Bewegung an einer anderen Position zu wiederholen.

- ∇ Es gibt keine *Generalisierung* oder Abstraktion einer Bewegung.
  Von Menschen wissen wir aber, daß sie Bewegungskonzepte (wie z.B. die Handschrift) auch mit anderen Bewegungsprimitiven durchführen können, wenn beispielsweise anstatt auf Papier klein mit Bleistift auf eine Tafel groß mit Kreide geschrieben wird. Dies ist bei dem beschriebenen Ansatz aber prinzipiell nicht möglich.

Zusammenfassend kann gesagt werden, daß die topologie-erhaltender Abbildungen einen interessanten Ansatz zur Roboterkontrolle auf neuronaler Ebene mit vielen interessanten Eigenschaften darstellen. Trotzdem gibt es aber noch einige wichtige Probleme für eine zufriedenstellende Theorie der Roboterkontrolle zu lösen.

4.3 Fehlertoleranz mit neuronalen Netzen

In den folgenden drei Abschnitten sollen die für den Gesichtspunkt der Fehlertoleranz wichtigen Aspekte sowohl der funktionellen als auch der Hardware- Fehlertoleranz sowie der Fehlerdiagnose mit neuronalen Netzen behandelt werden.

Im ersten Abschnitt werden Schwierigkeiten im Umgang mit Computern auf mangelnde funktionelle Fehlertoleranz der Benutzerschnittstelle zurückgeführt und die Notwendigkeit von selbstlernenden statt programmierbaren Rechnern gezeigt.

Im zweiten Abschnitt wird gezeigt, wo Fehlertoleranz bei dem neuronalen bzw. konnektionistischen Ansatz möglich ist. Nach Überlegungen zur Hardware-Fehlertoleranz wird gezeigt, daß die Probleme von "intelligenten", fehlertoleranten Aktionen auch bei der "Künstlichen Intelligenz" in den Bereichen Bilderkennung, Spracherkennung und Robotik ebenfalls vorhanden sind und sich auf drei grundsätzliche Problemkreise zurückführen lassen. Gute Lösungsansätze existieren auf dem Gebiet der neuronalen Netze, die deren *inhärente Fehlertoleranz* ausnutzen. Sie sind allerdings noch nicht konsistent und vollständig.

Im Abschnitt danach werden die Fehlertoleranzmöglichkeiten der neuronalen Netze am Beispiel des assoziativen Speichers näher untersucht. Der assoziative Speicher ist nicht nur ein wichtiges Beispiel für die Vorteile und Probleme der parallelen Funktion von kleinen, einzelnen Prozessorelementen, er hat auch interessante Fehlertoleranzproportionen. Es stellt sich heraus, daß durch Einführung von Nichtlinearitäten (z.B. Schwellwerten, s. Abschnitt 4.1) bei den Zustandsüberführungsfunktionen der einzelnen Prozessorelemente (Neuronen) sowohl ungenaue, fehlerhafte Eingabedaten als auch fehlerhafte Hardware toleriert werden kann. Beide Eigenschaften sind allerdings voneinander abhängig: Bei korrekten Daten können mehr Hardwareelemente defekt sein, ohne zu einer Fehlfunktion zu führen, und umgekehrt.

Im vierten Abschnitt wird kurz ein Ausblick auf die Anwendungsmöglichkeiten der neuronalen Netze bei der Diagnose von Systemzuständen (Fehlerdiagnose) gegeben.

4.3.1 Fehlertoleranz in intelligenten Benutzerschnittstellen

Die Integration der Computer in der Verwaltung, in Fabriken und Banken ist weniger ein Problem von leistungsfähigen Algorithmen, Datenbanken oder Maschinen, sondern das Problem, Menschen ohne "Computerverständnis" den Umgang mit den automatischen Daten- und Fabrikationssystemen zu ermöglichen. Das Grundproblem besteht darin, wie man "den Computer menschengerecht" machen kann, wie also die Benutzerschnittstelle der automatischen Systeme den Fähigkeiten und Gewohnheiten der Menschen angepaßt werden kann.

Eingabe-Fehlertoleranz *und menschengerechte Benutzerschnittstellen*

Betrachten wir die gewohnten menschlichen Informations-Eindrücke wie visuelle Wahrnehmungen (z.B. von Gesten) und akustische Wahrnehmungen (z.B. von Sprache), so bemerken wir, daß das reiche Spektrum menschlicher Ausdrucksfähigkeit geprägt ist durch Ergänzungen im Kontext: Sei es, daß wir durch nonverbale Gestik oder durch verbale Kurzsätze unseren Willen kundtun; stets muß der Zuhörer Informationen aus dem sprachlichen oder inhaltlichen Kontext ergänzen sowie falsche und unpassende Formen korrigieren, um eine sinnvolle Aussage oder eine sinnvolle Handlung zu erkennen. Diese menschliche Fähigkeit, beim Zuhörer (Empfänger) fehlertolerant Informationen zu abstrahieren, ermöglicht es wiederum dem Sprecher (Sender), sich mehrdeutig, fehlerhaft und unvollständig zu artikulieren und trotzdem noch verstanden zu werden.

So menschengerecht und problemlos diese Kommunikationsgewohnheiten bei Menschen sind, so problematisch ist dies bei Computersystemen. Aus frustrierenden Erfahrungen mit falschen oder fehlenden Computeraktionen heraus werden heutzutage deshalb nur eindeutige, logisch konsistente Befehle und Anfragen bei Computern zugelassen. Dazu muß der Benutzer eine spezielle, manchmal "benutzerfreundlich" genannte Sprache lernen, in der er mit dem System kommunizieren darf. Im Prinzip paßt sich damit der Benutzer im Denken dem Computer an, nicht umgekehrt.

Benötigt wird eine fehlertolerante bzw. fehlervermeidende Benutzerschnittstelle, die nicht nur formale (z.B. syntaktische) Fehler erkennt und korrigiert, sondern auch inhaltlich aus dem Kontext erkennt, was der Benutzer meint.

Fehlende Programmierung

Die Forderung nach einer fehlertoleranten Benutzerschnittstelle hat allerdings ein Problem: jeder Benutzer hat einen anderen Kontext, beispielsweise die Erfahrungen von mehreren Konsultationen des Computersystems. Eine gute Benutzerschnittstelle müßte also die Eigenheiten des Benutzers und seine Erfahrungen speichern, allerdings auf der abstrakten Ebene von Gewohnheiten. Beim Benutzen wird dabei die Schnittstelle klüger, sie *lernt* aus den Daten. Da für gute Benutzerschnittstellen bereits Expertensysteme eingesetzt werden, zielt die Forderungen nach Lernen dabei auf ein Grundproblem aller existierenden Expertensysteme. Im Unterschied zum menschlichen Experten, der beim Erstellen seiner Expertisen auch lernt (z.B. aus seinen Fehlern), müssen Expertensysteme immer wieder aktualisiert und von menschlichen Experten an das menschliche Wissen angepaßt werden. Normale menschliche Erfahrungen, aus denen menschliche Experten im Zweifelsfall ihre Analogien herleiten, sind ihnen prinzipiell verschlossen. Betrachten wir den Aufbau und die Pflege der Wissensbank eines Expertensystems als eine der effektivsten Formen der Programmierung heutzutage, so läßt sich das derzeitige Problem von fehlenden Programmierern in Industrie und Wirtschaft als typisch für die heutige Rechnerarchitektur betrachten: Sie ist nicht selbstprogrammierend bzw. selbstlernend. Gesucht ist eine Soft- und Hardwarearchitektur, die komplexe Fähigkeiten selbst lernen kann.

4.3.2 Fehlertolerante Neurocomputer

Was können neuronale Netze für die Lösung der bisher geschilderten Fehlertoleranzproblematik beitragen?

Betrachten wir zunächst das Problem der Fehlertoleranz gegenüber Ausfällen. Bei dem massiven parallelen Einsatz von Hardware steigt auch die Ausfallwahrscheinlichkeit des Gesamtsystems. Dabei stellen sich folgende Fragen:

Wie läßt sich erkennen, daß eine Einheit ausgefallen ist?
Wie werden Test, Diagnose und Reparatur (Rekonfiguration) durchgeführt?
Wie werden die durch Hardwaredefekte verfälschten Daten wieder restauriert?

Besonders die letzte Frage ist ziemlich heikel. Verwendet man ein Mehrheitsvotum von Einheiten (maskierende Fehlerkorrektur, s. Abschnitt 1.4), um umständliches Sichern von Zwischenergebnissen zu vermeiden (Rollback), so ist der Ausfall einzelner, votierender Einheiten beim Ergebnis nicht festzustellen. Dies kompliziert die Diagnosesituation. Auch zusätzlich eingebaute Testsignalpfade sagen nur etwas über Defekte in diesen Pfaden aus, nicht aber über den maskierten Datenpfad. Verwendet man andererseits keine Fehlererkennung, so ist es nur eine Frage der Zeit, bis die Mehrheit der votierenen Einheiten defekt ist.

Im Gegensatz dazu bereitet es uns Menschen keinerlei Probleme, daß in unserem relativ langsam und asynchron arbeitenden Gehirn (Schaltfrequenzen um den Faktor 10^4 geringer als beim Mikroprozessor) laufend informationsverarbeitende Nervenzellen bei der Arbeit wegsterben, ohne ersetzt zu werden. Da diese Eigenschaften ohne die Parallelität der Aktivität der Nervenzellen undenkbar sind, waren die Gehirntheoretiker - im Unterschied zu den Informatikern - schon seit Jahrzehnten gezwungen, Funktionsmodelle paralleler, fehlertoleranter Systeme zu entwickeln.

Konnektionistische Ansätze

Bei den parallelen Funktionsmodellen lassen sich zwei verschiedene Ansätze unterscheiden: der "lokalistische" und der " distributionistische" Ansatz.

Die symbolistischen oder *lokalistischen* Modelle erklären die Gesamtfunktion des Systems aus der Verbindung von Einheiten; jeder Verbindung korrespondiert dabei eine Einzelfunktion [FELD]. Die Verbindung von genau definierten Einzelfunktionen zu einer Gesamtfunktion liegt im Wesentlichen den schnellen Maschinen der AI [HILL] und den verteilten Algorithmen [SHA] zu Grunde.

Allerdings sind die Modelle dieses Ansatzes nicht besonders fehlertolerant: Fällt eine Einheit oder Verbindung aus, so ist damit auch eine Relation oder ein Objekt "vergessen". Diese Art von Netzwerken ist zwar geeignet, bestimmte Probleme schnell zu lösen, erlaubt aber Fehlertoleranz auf Hardware- und Softwareebene nur mit zusätzlichem, nicht unerheblichem Aufwand. In nicht-zentralisierten, verteilten Systemen ist ein Overhead zur Verwaltung der Redundanz (Systemzustandstafeln), Fehlererkennung, -diagnose, und Rekonfiguration nötig [KUHL]. Allein das Problem der Erkennung fehlerhafter Einheiten (Diagnose) ist bei funktions-verfälschenden Einheiten nicht einfach zu lösen [PEAR]. Das lokalistische konnektionistische Modell entspricht damit dem heutigen Ansatz, möglichst viele von-Neumann Maschinen zur parallelen Informationsverarbeitung

in einem Netz zusammen zu schließen. Leistungsverluste durch Hard- und Softwarezusätze für den Netzbetrieb (Netzwerkprotokolle) werden dabei in Kauf genommen und es wird versucht, sie durch günstige funktionelle Partitionierung des Problems zu minimieren.

Anders dagegen der Ansatz der *distributiven* Modelle. Sie repräsentieren die Aktivitätsmuster nicht lokal in einer Verbindung, sondern als gemeinsame Aktivität aller Verbindungen. Die Gesamtheit aller Verbindungen läßt sich mittels einer Verbindungsmatrix beschreiben, so daß die Gesamtfunktion als Wechselwirkung aller Neuronen untereinander mit einer gewichteten Verbindungsmatrix modelliert wird [KOH1]. Aus der Holografie ist bekannt, daß die Beschädigung von Teilen der Bilderfolien bei der Reproduktion des Bildes nur die Qualität des Gesamtbildes herabsetzt, nicht aber Bildteile verschwinden läßt. Dies ergibt sich aus der Tatsache, daß die Bildinformationen nicht lokalisiert, sondern prinzipiell an jedem Punkt der Bildfolie vorhanden sind. Da die Matrix-Modelle ähnliche Proportionen aufweisen, wurden sie in Analogie zum physikalischen Pendant anfangs "Hololologische Modelle" genannt [LONG], [WIL].

Neuere Arbeiten zeigen allerdings verschiedene Wege, beide Ansätze miteinander verschmelzen zu lassen. Beispielsweise führten Sohn und Gaudiot [SOHN] assoziative Speicher ein, um Produktionssysteme parallel zu modellieren. Hierbei ist eine Symbolzuordnung zu einem Signal äquivalent zu einem Mustervektor, der in einer Komponente aktiv ist. Der prinzipielle Gegensatz in der Fehlertoleranz zwischen dem lokalistischen und dem distributiven Ansatz wird damit zu einem graduellen Unterschied zwischen zwei verschiedenen Kodierungsschemata. Die zweite Ereigniskodierung ist nicht absolut besser, sondern nur in dem durch die Kodierung bestimmten Maße. Den Einfluß der Kodierung auf die Fehlertoleranz in assoziativen Speichern ist in [BR11] näher untersucht worden.

Außer diesen fehlertoleranten Hardwaremechanismen zur assoziativen Speicherung von Informationen, auf die noch in Abschnitt 4.3.3 näher eingegangen werden wird, sind allerdings noch andere Probleme zu lösen, um eine fehlertolerante Maschine zu bauen, die eine fehlertolerante, intelligente, selbstlernende Benutzerschnittstelle verwirklichen könnte.

Dazu gehört beispielsweise *intelligentes* Verhalten. Um nicht auf die psychologischen, sozialen, genetischen u.ä. Aspekte eingehen zu müssen, beschränken wir uns bei dieser Betrachtung auf die höheren Leistungen, die in dem Fachgebiet der "künstlichen Intelligenz" damit verbunden werden. Es gibt interessante Ansätze, konnektivistische Modelle in den Gebieten Bilderkennung, Sprachverarbeitung und Robotik der künstlichen Intelligenz einzusetzen. Dabei ist die Eigenschaft *Fehlertoleranz* untrennbar verbunden mit den grundlegenden Funktionen dieser Modelle; *Datenkorrektur, Abstraktion, Mustererkennung* und *Kategorisierung* sind inhärente Fehlertoleranzmechanismen. Damit überdeckt sich die Untersuchung der Fehlertoleranzeigenschaften der konnektivistischen Modelle weitgehend mit der Untersuchung ihrer Grundfunktionen.
Betrachten wir nun die Probleme der drei genannten Gebiete genauer.

Bildverarbeitung

Abgesehen von der Forderung, möglichst alle Punkte eines Bildes (Pixel) gleichzeitig zu bearbeiten, ist die Bildverarbeitung üblicherweise in verschiedene Stufen oder Schichten aufgeteilt, wie sie in Abbildung 4.2.1 gezeigt sind.

Lassen sich Bilderfassung und -verbesserung noch mit klaren, nachrichtentechnischen

Algorithmen durchführen, so stellt schon die Erkennung eines Umrisses (*Segmentierung*) ein Problem dar. Viele Umrisse sind durch Störungen und Verdeckungen nicht vollständig. Eine der besten Techniken, um kleine Lücken zu schließen, besteht in der Ergänzung dieser Stellen mittels der Information von Nachbarpixeln (*Relaxation*). Ist dagegen eine starke Oberflächentextur vorhanden, so versagen auch diese Methoden und es muß, am Besten mittels stochastischer Mustererkennungsverfahren, die Flächen mit gleicher Textur erkannt und abgegrenzt werden [BALL].

Auch beim Erkennen von Gesamtobjekten und Situationen, die im allgemeinen Fall mit wissensbasierten Methoden (semantischen Netzen u.ä.) behandelt werden, kämpft die Objekterkennung bei falscher Identifizierung von Teilobjekten (Grund: Störungen auf unterer Ebene) und unvollständigen Objekten mit dem gleichen Problem: Bildmuster, die den gespeicherten Referenzmustern ähnlich, aber nicht gleich sind, müssen dem ähnlichsten Referenzmuster zugeordnet werden.

Dies gilt übrigens auch bei der Erkennung von zeitlichen Sequenzen von Bildern (Bildfolgen): Menschen und Gegenstände, die sich bei der Bewegung leicht ändern, müssen trotz Variation und Verschiebung auf jedem Bild der Folge wiedererkannt werden.

Spracherkennung

Auch in der Spracherkennung haben sich die mehrstufigen Verfahren und Modelle ebenso wie in der Experimentalpsychologie durchgesetzt. Ein solches Schichtenmodell ist beispielsweise in Abbildung 4.2.1 gezeigt.

Reichte die Methode, eine Lautcharakteristik (zeitliche, endliche Folge von Amplituden- und Frequenzwerten) mittels Durchsuchen einer Liste von bekannten Lauten und Identifizierung mit einem Ähnlichkeitskriterium (*Dynamic Time Warping*) noch aus, Ein-Wort-Systeme mit einem Wortschatz von ca. 100 Worten zu bauen, so stößt diese Ein-Schritt Methode bei Sätzen schon an ihre Grenzen: In normalen Texten gibt es keine zwei gleichen Sätze, da die Kombinationsmöglichkeiten von Einzelworten zu groß sind. Gute mehrstufige Systeme versuchen, die regionalen und individuellen Sprachunterschiede auf Laut-, Wort- und Satzebene durch Systeme mit Wahrscheinlichkeitsaussagen zu modellieren (*Hidden Marcov Models)* und diejenigen Übergänge zwischen Spracheinheiten (Laute, Worte) zu finden, die am wahrscheinlichsten mit den unbekannten Lautsequenzen übereinstimmen [DE MORI].

Die Probleme, die diese Ansätze mit sich bringen, ähneln sehr denen der Bildverarbeitung:

- Es müssen Listen von gespeicherten, ähnlichen Worten durchsucht werden, um das Passende zu finden
- Variationen bzw. Fehler und fehlende Laute können zu Fehlern führen; beim Schichtenmodell der *hidden marcov models* müssen die Wahrscheinlichkeiten auf verschiedenen Ebenen gleichzeitig berücksichtigt werden, um das im Kontext Passende zu finden. Dabei sind aber auch die Übergangswahrscheinlichkeiten der Lautmuster individuell vorherzubestimmen (starker Rechenaufwand).

Betrachtet man im Gegensatz dazu das menschliche Vermögen, Sprache zu erkennen, so zeigen sich die grundlegenden Unterschiede zwischen beiden Ansätzen. Schon auf Lautebene (*Phoneme*) registriert jeder Computer ein Kontinuum an physikalischen Lautformen; die menschliche

Sprachverarbeitung aber läßt uns alle kleineren Lautvariationen als ein- und denselben Laut hören, sofern eine bestimmte Variationsgrenze nicht überschritten wird (*Kategoriale Sprachwahrnehmung*). Diese Kategorisierung ist eine Musterkennung und setzt sich auf höheren Stufen fort, so daß die Fehlertoleranz bezüglich der Eingabedaten als wichtiges Funktionsprinzip unserer Sprachwahrnehmung auf allen Ebenen realisiert sein muß.
Umgekehrt ergibt sich daraus aber auch der große Erfolg beim Computereinsatz zur Sprecheridentifizierung (z.B. Erpresserstimmen am Telefon), wo Menschen ziemlich versagen, da sie nicht exakt die Lautform, sondern nur die Kategorien bewußt wahrnehmen.

Robotik

In der Robotik gelten die gleichen, vorher erläuterten Probleme bei der Sensorik (Bilderkennung, Spracherkennung, taktile Sensoren). Zusätzlich sind hier Aktivierungssignale nötig, um die Gliedmaßen zu steuern. Diese Steuerung hat einige Probleme zu lösen. Um ein bestimmtes Ziel mit dem Greifer zu erreichen, müssen alle Segmente und Gelenke des Roboterarmes gleichzeitig mit der richtigen Beschleunigung und Geschwindigkeit um den richtigen Betrag bewegt werden.
Dies erfordert

- Umrechnung der Gesamtbewegung in Einzelbewegungen innerhalb der gelenkspezifischen Koordinatensysteme (*Inverse homogene Transformationen*).
  Dabei können Probleme auftreten (Singularitäten).
- Einbeziehung der Massen und Bewegungsenergien der Einzelsegmente und der aufgenommenen Last in die Bewegungsgleichungen der Gelenksteuerungen.

Dazu sind sowohl Lösungen von Matrizengleichungen als auch Differenzialgleichungen zweiter Ordnung nötig. Werden zusätzliche Forderungen an die Bewegung gestellt (*Konstante Kraft; gleichförmige, lineare Bewegung im Kartesischen Raum*), so müssen außerdem die Daten der Positions- und Kraftsensoren ausgewertet werden. Die bei der ungenauen Bewegung (Schlupf der Kraftransmission, Durchbiegen des Arms bei schweren Lasten, Bewegungshindernis) nötige Fehlertoleranz wird meist durch eine Kombination von Rückkopplung der Sensorsignale und einer Liste von Ausnahmesituationen ("Werkstück heruntergefallen") verwirklicht.
Probleme macht bei diesem Ansatz der Bewegungssteuerung neben der Berechnung diverser Gleichungen in Real-Time die Programmierung dieser Bewegungen. Die gewünschte Bewegungssequenz wird entweder in einer Programmiersprache eingegeben (z.B. *moveTo (x,y,z)*) oder, soweit möglich, direkt mittels der menschlichen Bewegung eines Testarms eingegeben (*Playback robots*) [FU].

"Intelligente" Roboter, die selbst neue Bewegungen auf Grund äußerer Sensordaten vollführen können (z.B. Kollisionsvermeidung) gibt es praktisch ebensowenig wie Roboter, die nicht nur einen, sondern zwei oder mehr Arme oder Beine koordiniert bewegen können, sogar noch durch Bildauswertung gesteuert. Bei all diesen Problemen trifft die von-Neumann Architektur auf ihre Grenzen.

Neuere Versuche, diese Grenzen zu überwinden, bauen auf Softwaresystemen aus hierarchischen Schichten auf. Jede Schicht ist ähnlich gebaut: Sie empfängt Befehle von oben und setzt sie in konkrete Handlungssequenzen für die untergeordneten Schichten um. Andererseits wer-

den die Sensorsignale der tieferen Schichten bei der Erstellung der Handlungssequenzen verwendet (*feed-back*) sowie in abstrakterer, codierter Form nach oben weitergereicht (Beispiel: digitale Positionsdaten werden zur Meldung "Greifer im kritischen Bereich"). In der Abbildung 4.2.8 in Abschnitt 4.2 ist solch ein Schichtenmodell gezeigt; parallel dazu rechts in der Abbildung der vermutliche Aufbau der menschlichen Motorik (nach [SCH]).
Allgemein läßt sich feststellen, daß für die Bewegungssteuerung der Roboter eine Hard- und Softwarearchitektur benötigt wird, die nicht explizit programmiert zu werden braucht, sondern selbst die Bewegungsprimitive lernen kann und daraus die benötigten Bewegungen synthetisiert.

Neuronale, fehlertolerante Computer

In den vorigen Abschnitten wurden die allgemeinen und speziellen Probleme der fehlenden Fehlertoleranz bezüglich Eingabefehler in der Sensorik (Intelligente, menschengerechte Benutzerschnittstellen) und der schwierigen oder fehlenden Programmierung (Lernende Schnittstellen und Roboter) vorgestellt. Alle vorgestellten Probleme lassen sich in Schichtenmodellen beschreiben, wobei jede dieser einzelnen Schichten unter anderem folgende Fehlertoleranz-Eigenschaften haben sollte:

- unempfindlich gegenüber Störungen der Eingabedaten
- Ergänzung unvollständiger Daten
- Kategorisierung (Klassifizierung, Mustererkennung) der Daten
- Selbstlernen der Kategorien und Assoziationen

Abstrahieren wir von den konkreten Anwendungen der Künstlichen Intelligenz, so lassen sich im wesentlichen drei Problemkreise abgrenzen, deren Lösung eine zentrale Bedeutung für die Bewältigung obengenannter Probleme und damit für die Konstruktion der neuronalen Computer zukommt. Beschreiben wir die Bild-, Ton- und Aktivierungsereignisse durch eine Menge von Variablen, zusammengefaßt wie in Abschnitt 4.1 in einem Mustervektor **x**, so werden folgende Operationen für diese Muster benötigt:

1) **Feature extraction**

   Aus unbekannten Sensordaten (Input-Muster) müssen die *typischen, charakteristischen Merkmale* als solche erkannt und extrahiert werden. Leider ist der Begriff "typisches Merkmal" nicht sehr präzise definiert. Deshalb gibt es viele Modelle, die vorgeben, eine Merkmalsextraktion durchzuführen und in Wirklichkeit nur eine Form von Mustererkennung durchführen. Zu einer echten Merkmalsextraktion gehört dagegen das Erkennen und Auswählen von Merkmalen, die vorher unbekannt waren; also die Ermittlung von Zahl und Art von unbekannten Merkmalen und eine Rekodierung ("Abstraktion") der Originalinformation, was bisher in den Systemen der symbolischen, künstlichen Intelligenz nicht möglich ist.
   Ein interessanter Lösungsansatz dazu sind die "optimalen Schichten" aus Abschnitt 4.1. Die Ausgabecodes sind hier die Anteile der Eigenvektoren am eingegebenen Muster (Eigen vektor-Dekomposition). Weitere geeignete Modelle bilden beispielsweise die Gruppe der stochastisch selbstlernenden Konkurrenz-Modelle (*Competetive Learning*, z.B. [GROS] oder Kohonens *topologie-erhaltenden Abbildungen*, siehe Abschnitt 4.2).

2) Zeitsequenzen

Im Unterschied zur statischen Bildverarbeitung, bei der alle Bildmerkmale gleichzeitig vorliegen, ist bei der Bildfolgeerkennung und bei den zeitlichen Sequenzen der Sprachlaute das zu erkennende Muster zeitlich auseinandergebrochen. Gesucht ist ein Mechanismus, der die Operationen auch bei zeitlich sequentiellen Mustern durchführt.

Auch die Aktivierungsmuster der Roboter sind zeitlich gegliedert. Beide Probleme hängen eng zusammen: Ist ein befriedigender Mechanismus gefunden, ein zeitlich sequentielles Muster fehlertolerant zu erkennen und den dazu assoziierten Code auszugeben, so wird auch bei autoassoziativem Gebrauch durch Eingabe des Codes mittels Ergänzung die entsprechende zeitliche Sequenz ausgelöst.

Leider gibt es nur wenige Modelle, die Zeitsequenzen darstellen können. Vielfach tritt auch das Problem auf, daß zwei Folgen, die identische Teilfolgen enthalten, bei der Generation bzw. Erkennung verwechselt werden können.

3) Assoziativer Speicher

Funktionen des assoziativen Speichers sind

- Speichern der Referenzmuster
- Vergleich unbekannter Muster mit den gespeicherten Referenzmustern, Entscheidung auf das ähnlichste Referenzmuster

Die Modelle für Assoziativspeicher haben eine lange Tradition und sind sehr zahlreich. Allgemein läßt sich feststellen, daß diese Assoziativspeicher für ihre Speicherfunktion weniger Muster speichern können (je nach Modell 15%-60%) als die konventionellen adress-orientierten Speicher und Assoziativspeicher mit flag-Architektur. Dafür aber sind die verteilten Assoziativspeicher schneller (die Speicher- und Ausleseoperation ist in einem Funktionszyklus beendet) und, wie wir im nächsten Abschnitt sehen werden, bei nicht-linearer Ausgabefunktion auch fehlertolerant.

Mit diesen drei Grundoperationen lassen sich die untersten Schichten der vielschichtigen Bild- und Spracherkennung konstruieren: Selbstprogrammierung und Lernen sowie Codierung der Sensorereignisse mit Operation 1; Bild- Spracherkennung und taktile Erkennung sowie Robotersteuerung mit Operation 1, kombiniert mit Operation 2. Alle Sensorprimitive (z.B. Eigenvektoren) werden dabei mit Operation 3 gespeichert und abgerufen.

Die drei Operationen sind sicher nicht unabhängig voneinander: Grundlegend ist das Speichern von Referenzmustern und der schnelle, fehlertolerante Vergleich mit unbekanntem Input durch assoziativen Abruf. Versehen wir den Assoziativspeicher mit einem Kurzzeitspeicher, so läßt sich Operation 2 konstruieren (vgl [KOH3]). Auch bei der Merkmalsgewinnung der Operation 1 wird die Grundfunktion des Speicherns und Vergleichens der Merkmale benötigt.

Obwohl viele Modelle Berührungspunkte haben und ähnliche Phänomene erklären wollen, gibt es leider noch kein einheitliches Theoriegerüst für die Neuronalen Netze, ebensowenig wie umfassende Untersuchungen ihrer (zweifellos vorhandenen) Fehlertoleranzeigenschaften. Es ist noch viel Entwicklungsarbeit zu leisten, um alle Probleme zu lösen, die bei diesem speziellen Ansatz auftreten. Beispielsweise bedeutet ein Schritt zu Computern mit menschenähnlichen Leistungen ja auch, die Fehler zuzulassen, die Menschen normalerweise dabei machen.

4.3.3 Fehlertoleranz in Nichtlinearen Neuronalen Netzen

Im Folgenden wollen wir genauer untersuchen, was die Einführung von Schwellwerten (Nichtlinearitäten) bei linear gekoppelten und zu einem Assoziativspeicher geschalteten Verarbeitungseinheiten (formale Neuronen) für die Tolerierung von defekten, fehlerhaften Eingabemustern beim Assoziativspeicher bewirkt.

Es ist allgemein bekannt, daß Schwellwerte bei Signalpegeln zur Rausch- und Störunterdrückung und damit zur Fehlertoleranz bezüglich der Signalpegel beitragen. Zwar wurden die potentiellen Fehlertoleranzmöglichkeiten ansatzweise erkannt (z.B. Grossberg [GROS],p.125; Kohonen [KOH3]p.165) oder in der Simulation gefunden [BEL], aber nicht ausführlich untersucht. Da die Fehlertoleranz-Eigenschaften ohne zusätzliche Maßnahmen und ausschließlich aus den funktionellen Proportionen des Modells resultieren, wird dies als *inhärente Fehlertoleranz* bezeichnet.

Betrachten wir dazu das Modell des verteilten, assoziativen Speichers etwas näher.

Der lineare, assoziative Speicher

Seien die Eingabemuster (Ereignisse, Reize) durch einen reellen Vektor $\mathbf{x} = (x_1,...,x_n)$ und die dazu assoziierten Ausgabemuster durch reelle $\mathbf{y}=(y_1,...,y_m)$ beschrieben (s. Abschnitt 4.1), so läßt sich die Verknüpfung zwischen beiden Mustern linear mittels der Matrix $\mathbf{W} = (w_{ij})$ modellieren:

$$\mathbf{y} = \mathbf{W}\,\mathbf{x} \qquad \text{mit der Transferfunktion } T_i(\mathbf{x},t) = \mathbf{w}_i\mathbf{x}$$

In einer Implementierung entspricht den Matrixkoeffizient w_{ij} hardwaremäßig die "Stärke" der Verbindung zwischen der Eingabeleitungen x_j und der Ausgabeleitung y_i, s. Abbildung 4.3.1.

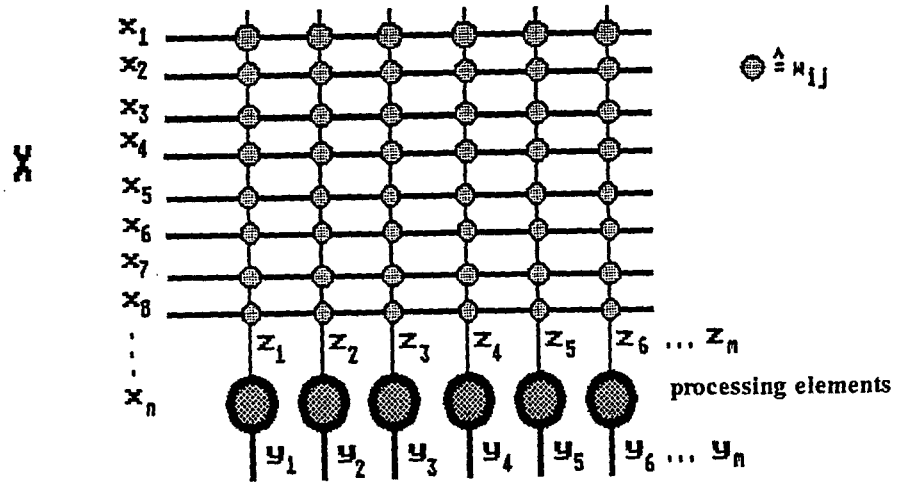


Abb. 4.3.1 Ein Hardwaremodell eines assoziativen Speichers

Die Aktivitäten der einzelnen Komponenten von x summieren sich gewichtet in z_i (z.B. als elektr. Ströme) und erzeugen ein Ausgabesignal in y_i.

Zum Speichern eines Paares (x,y) werden gleichzeitig x und y an den Ein- bzw. Ausgängen präsentiert und die Gewichte an den Kreuzungspunkten verändert, beispielsweise nach der

Hebb'schen Regel

$$w_{ij} \sim y_i x_j \qquad \textit{Hebb'sche Regel}$$

Nach dem Anlegen von p Mustern $x^1,...,x^p$ resultieren die Gewichte

$$w_{ij} = \sum_k c_k y_i^k x_j^k$$

mit der Proportionalitätskonstanten c_k.
Wird diesem System ein bereits gespeicherter Vektor x^r erneut präsentiert, so ergibt sich als Ausgabe

$$y = W x = z = c_r y^r x^r x^r + \sum_{k\#r} c_k y^k x^r x^k \qquad (4.3.1)$$

ass.Antwort + Übersprechen von anderen Mustern

Normieren wir die Gewichte mit $c_k := 1/(x^k x^k)$ [mit der Notation **vw** für das innere Produkt zweier Vektoren **v** und **w**] und verwenden ein System von orthogonalen Speichervektoren x^k, so ergibt sich mit $x^i x^j = 0$, i#j wieder die zu x^r assoziierte Antwort y^r.

Fehlerhafte Eingabe
Was geschieht nun, wenn wir ein von x^r abweichendes Muster **x** dem linearen System präsentieren ? Sei die Abweichung zu **x** mit $x' := x - x^r$ bezeichnet, so resultiert als Ausgabe

$$y = W x = W (x^r + x') =: y^r + y'$$

Orignal Störterm

die Überlagerung aus der zu x^r assoziierten Antwort und einem "Störterm", der aus einer Linearkombination aller y^k gebildet wird.

Da bei der Ausgabe nur die stärkste Komponente gefragt ist, kann man die korrekte Ausgabe automatisch dadurch erhalten, daß man alle Komponenten von **y** vor der Ausgabe einer Schwellwertoperation unterwirft. Ist die Schwelle geeignet gewählt, so wird nur eine Komponente sie überschreiten und das korrekte y^r produzieren. Dies ist die Grundidee des linearen Assoziativspeichers mit Schwellwert.

Der lineare Speicher mit Schwellwert

Bisher betrachteten wir reelle Vektoren **x** und **y**. Identifizieren wir die reellen Werte mit den Spikefrequenzen der Neuronen, so beschränkt sich der Wertebereich der **x** und **y** auf positive Zahlen, da keine negativen Spikefrequenzen existieren.
Betrachten wir nun den Fall beliebiger x^k und orthogonaler y^k (orthogonale Projektion der **x** auf **y**), so gibt es nur ein y^{k_i}, bei dem die Komponente y_i ungleich null ist. Damit vereinfacht sich die Gleichung (4.3.1) zu

$$z_i = y_i^{k_i} c_{k_i} x x^{k_i} \qquad \text{mit } k_i \text{ aus } 1..p,\ i \text{ aus } 1..m \qquad (4.3.2)$$

Verwenden wir beliebige, und nicht wie Kohonen, orthogonale x^k, so resultiert als Summenvektor **z** ein Vektor, der in jeder Komponente z_i das innere Produkt aus dem Eingabevektor **x** und dem einzigen Speichervektor x^{k_i}, der zu **y** mit der Komponente y_i ungleich null assoziiert wurde.
Damit ist das Ausgabemuster eine Funktion des Vektorprodukts, das die Kreuzkorrelation zwischen

Inputmuster $\mathbf{x}$ und einem der Speichermuster $\mathbf{x}^{ki}$ darstellt:

$$y_i = f(z_i) = f(\mathbf{x}\mathbf{x}^{ki})$$

Welche Anforderungen werden an die Funktion $f(z_i)$ gestellt?

Maximale Korrelation

Bei einem Eingabemuster $\mathbf{x}$, das dem gespeicherten Muster $\mathbf{x}^{ki}$ besonders ähnlich ist, ist die Kreuzkorrelation $\mathbf{x}\mathbf{x}^{ki}$ besonders groß. Um bei Eingabe von $\mathbf{x}^{ki}$ eine Ausgabe von y_i^{ki} zu bewirken, muß die Funktion $f_i(.)$ bei besonders hoher Kreuzkorrelation die Komponente y_i^{ki} ausgeben und bei allen anderen Mustern $\mathbf{x}$ eine Null. Eine Ausgabe größer Null ist also nur bei einem Neuron r zu erwarten, das dem *Ähnlichkeitskriterium*

$$\mathbf{x}\mathbf{x}^r = \max_k \mathbf{x}\mathbf{x}^k \qquad (4.3.3)$$

genügt. Die Entscheidung für den Vektor $\mathbf{x}^{ki}$ in dem neuronalen Element i stellt somit eine *Mustererkennung* dar; die Menge aller auf $\mathbf{x}^{ki}$ abgebildeten $\mathbf{x}$ bildet eine *Klasse*, repräsentiert durch den *Klassenprototypen* $\mathbf{x}^{ki}$. Das Ähnlichkeitskriterium, mit dem über die Einordnung entschieden wird, ist die Korrelation mit dem Klassenprototypen [BR8].

Über die allgemeine lineare Separierung des Musterraums der Eingabemuster $\mathbf{x}$ hinaus, wie sie in Abschnitt 4.1 prinzipiell für Perzeptrons vorgestellt wurde, werden die Gewichte beim Assoziativspeicher derart durch die Hebb'sche Regel festgelegt, daß (bei orthogonaler Ausgangscodierung) jedes neuronale Element des assoziativen Speichers wie ein Klassifikator wirkt, der durch ein einziges Muster, den Klassenprototypen, festgelegt ist.

Wie ist nun die Klassifizierung durch solch eine spezielle Wahl der Gewichte charakterisiert?

Betrachten wir die geometrische Veranschaulichung dieser Mustererkennung. Für jedes Muster $\mathbf{x}$ wird diejenige Klasse r gewählt, bei der für alle anderen Klassenprototypen $\mathbf{x}^k$ gilt $\mathbf{x}\,\mathbf{x}^k < \mathbf{x}\,\mathbf{x}^r$.

Es wird in [BR8] gezeigt, daß dadurch erwartungsgemäß (vgl. Abschnitt 4.1) eine Hyperfläche als Klassengrenze definiert wird. Interessanterweise ist dies eine Ebene, die den Musterraum zwischen $\mathbf{x}^r$ und $\mathbf{x}^k$ auf halbem Wege zwischen beiden Endpunkten zerteilt. Damit ist eine Entscheidung für eine Klasse mit einem konstanten Schwellwert ohne explizite Kenntnis der anderen Klasse möglich.

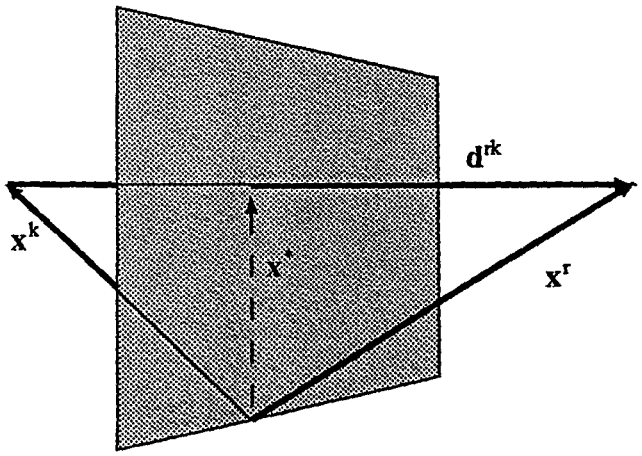


Abb. 4.3.3 Klassentrennung im 3-dim Musterraum

In Abbildung 4.3.3. ist eine geometrische Verdeutlichung im 3-dimensionalen Musterraum gezeigt.

Allerdings ist die korrekte Klassifikation mit dem Kriterium der maximalen Korrelation nur eingeschränkt möglich. Um die korrekte Erkennung des Klassenprototypen selbst (gespeichertes Muster) zu ermöglichen, sollten alle anderen Klassenprototypen nicht im Gebiet sein, das durch die Hyperebene (Gerade) orthogonal zu und durch $\mathbf{x}^r$ begrenzt wird. In der folgenden Abbildung sind die "verbotenen Gebiete" grauschraffiert für drei gespeicherte Muster (Klassenprototypen) im 2-dim Fall gezeigt.

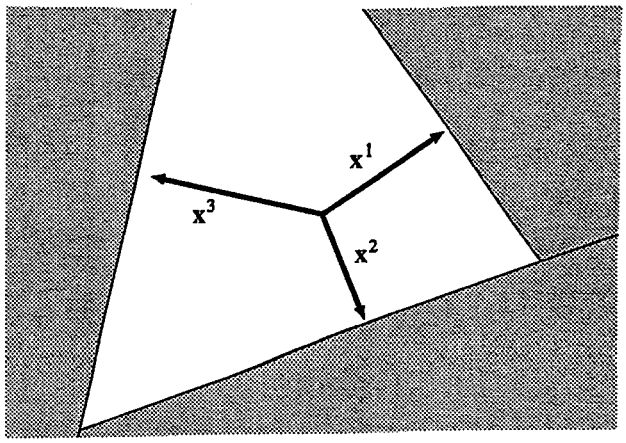


Abb.4.3.4 Restriktionen für die Speicherung von Mustern

Minimaler Abstand

Betrachten wir deshalb ein anderes Ähnlichkeitskriterium, das weniger Restriktionen verlangt. Mit der Forderung

$$|\mathbf{x}-\mathbf{x}^r| = \min_k |\mathbf{x}-\mathbf{x}^k| \qquad (4.3.4)$$

wird ein Klassifikationschema und damit eine Aufteilung des Musterraums in Klassen definiert. In [BR11] wird gezeigt, daß die Trennfläche ebenfalls eine Hyperfläche ist und den Differenzvektor $\mathbf{d}_{rk} = \mathbf{x}^r - \mathbf{x}^k$ ebenfalls bei halbem Abstand $\mathbf{d}_{rk}/2$ schneidet. Im Unterschied zum Kriterium der maximalen Korrelation ist allerdings die Hyperfläche jeweils orthogonal zu $\mathbf{d}_{rk}$ orientiert, siehe Abbildung.

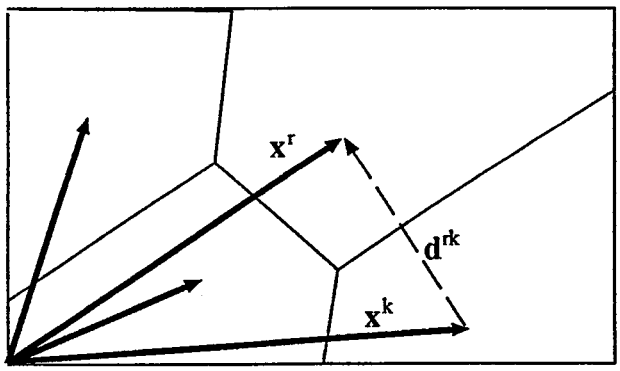


Abb.4.3.4 Aufteilung des Musterraums bei min. Abstand

Interessanterweise können die Klassenprototypen als Zustand der Gewichte der Neuronen in Kohonens *Topologie-erhaltenden Abbildungen* (s. Abschnitt 4.2.1) angesehen werden; die Auswahlregel (4.2.1) entspricht der Auswahlregel (4.3.4).
Vergleichen wir die beiden Abbildungen (4.3.3) und (4.3.4) miteinander, so sehen wir, daß die Klassenprototypen im zweiten Bild niemals mit Hilfe des Korrelationskriteriums korrekt erkannt werden können.
Was sind nun die optimalen Schwellwerte t_i für beide Ähnlichkeitskriterien?

Klassifizierung und Parallelarbeit

In den vorher vorgestellten geometrischen Illustrationen wurde demonstriert, wie die Klassifizierung durch die Lage der Klassengrenzen bestimmt wird. Diese Klassifikationen kann man mit Hilfe einer nicht-linearen Transferfunktion (siehe Abschnitt 4.1) eines formalen Neurons implementieren: Ist die Aktivität des Neurons groß genug ($t_i^r(k) < z_i$), so gehört das Muster zu der Klasse r; ansonsten ($t_i^r(k) \geq z_i$) liegt nur eine Störung vor.
Allerdings ergeben sich zwei wesentliche Probleme dabei.

- Die Ähnlichkeitsbedingungen (4.3.3) und (4.3.4) sind prinzipiell globale Betrachtungen. Jedes Prozessorelement (Neuron) müßte, um festzustellen ob es aktiv sein soll oder nicht, seinen Wert für $\mathbf{x}\mathbf{x}^r$ bzw. $|\mathbf{x}-\mathbf{x}^k|$ mit den Werten aller anderen Prozessoren vergleichen, was aber zu einer starken Kommunikation (Vernetzung) zwischen den Prozessoren führen und synchronisierte Operationen erfordern würde. Eine solche Vernetzung ist aber im Modell nicht vorgesehen.

- Versucht man die Klassifikation direkt durch unabhängige, parallele Schwellwertoperationen zu implementieren, so stößt man auf die Schwierigkeit, daß die Klassifikation nur die Unterscheidung zwischen den Klassen r und k realisiert. Für jede weitere Klasse müßte ein weiteres Neuron eingeführt werden, insgesamt also p(p-1)/2 Neuronen für p zu speichernde Muster.

Führt man stattdessen bei der Transferfunktion eine Schwelle ein, die allen Bedingungen gerecht wird, so wird die Schwelle von der stärksten Korrelation bzw. dem kleinsten Abstand zwischen den gespeicherten Mustern bestimmt. Dies führt aber zu einer Menge von Mustern, die mit der parallelen, dezentralen Entscheidung nur auf den Nullvektor (keine Klasse) abgebildet werden, obwohl ihre Klassenzugehörigkeit durch die globale Ähnlichkeitsbedingung eindeutig bestimmt ist.
Wir wir sehen, lassen sich mit einem Modell ohne Interprozessorkommunikation (im neuronalen Fall: "laterale Inhibition" oder ähnliche Kopplung) solche globale Klassifizierungskriterien nicht direkt implementieren. Stattdessen lassen sich nur Schwellen für hinreichende Klassifikation errechnen.
Ein hinreichender Schwellwert für die Unterdrückung der störenden Assoziationen (Übersprechen) ergibt sich in [BR8] für das Ähnlichkeitskriterium der maximalen Korrelation :

$$t_i = 1/2\,(K^r_{max} - |\mathbf{x}|^2) \qquad K^r_{max} := \max_k \mathbf{x}^r\mathbf{x}^k \qquad (4.3.5)$$

Für das Ähnlichkeitskriterium ergibt sich in [BR11] für das binäre Modell mit dem Hammingabstand $d_H(\mathbf{u},\mathbf{v})$ zweier Vektoren $\mathbf{u}$ und $\mathbf{v}$

$$d_H^r := \min_k d_H(\mathbf{x}^r,\mathbf{x}^k)$$

$$t_i^r := 1/2\,(\,|\mathbf{x}^r|^2 + |\mathbf{x}|^2 - d_H^r/2) \qquad (4.3.6)$$

Bei konstanter Aktivität der Eingabemuster (konstanter Länge $|x^k|^2 := a$) ist

$$K^r_{max} = 1/2(2a-d_H^r) = a - d_H^r/2$$

so daß sich die Korrelationsschwelle auch als Funktion des minimalen Hammingabstandes d_H^r der Klasse r zu seinen Nachbarn ausdrücken läßt in Übereinstimmung mit (4.3.5).
Hierbei ist die Klassengrenze bei $d_H^r/2$, ein Resultat, das gut mit der Kodierungstheorie (Fehlererkorrektur bei Blockcodes) übereinstimmt. Vorschläge zur Hardwareimplementierung der musterabhängigen Schwellen sind in [BR11] enthalten.

Geometrisch läßt sich die Klassifizierung mit dem kleinsten Abstand als Klassifizierung aller innerhalb eines Kreises mit dem Radius d^r liegenden Muster zum Prototypen verdeutlichen. Alle Punkte, die zwischen dem Kreisbogen und der Klassengrenze liegen, werden auf den Nullvektor abgebildet. Erweitert man die Kreisbögen, so daß sich alle Bögen überschneiden und jedes Muster mindestens innerhalb eines Kreises (einer Klassifizierung) liegt, so gibt es auch Muster, die gleichzeitig in zwei verschiedene Klassen eingeordnet werden. Die resultierenden Ausgabevektoren des Assoziativspeichers sind somit als Überlagerung leicht verändert und haben einen geringen, nichtverschwindenden Abstand zu den korrekten Ausgaben.

Inhärente Fehlertoleranz und Datenbanken

Zusammenfassend sahen wir in diesem Abschnitt, daß mit der Einführung einer Schwelle die Menge aller möglichen Eingabemuster in Untermengen (Klassen) zerfällt, die durch die gespeicherten Muster als Klassenprototypen festgelegt sind. Die Ausleseoperation des Assoziativspeichers wird damit zu einer Mustererkennungsoperation. Hierbei erregt das Eingabemuster dasjenige Ausgabemuster, das zu dem Klassenprototypen mit der größten Ähnlichkeit zum Eingabemuster assoziiert ist.

Die Tatsache, daß vom gespeicherten Muster abweichende Eingabemuster die selbe, korrekte Ausgabe bewirken, läßt sich auch als *Toleranz gegenüber fehlerhaften Daten* interpretieren. Die Vektorquantisierungseigenschaften dieses Modells bewirken somit nicht nur einen Mustererkennungs- oder Kategorisierungsprozeß, sondern sind auch mit einem *inhärenten Fehlertoleranzmechanismus* gegenüber gestörten Eingabedaten identisch.

Assoziiert man zu jedem Eingabemuster das selbe Muster als Ausgabe (Autoassoziativer Speicher), so bedeutet die Toleranz gegenüber fehlerhaften Daten eine Korrektur- und Ergänzungsoperation der Eingabedaten. Diesen Mechanismus der Datenergänzung von unvollständigen Daten kann man dazu verwenden, Tupel von Daten, beispielsweise Relationen zwischen zwei Objekten *(Relation, Objekt1, Objekt2)* zu speichern. Ein Auslesen bzw. Ergänzen eines unvollständigen Tupels *(Relation, - , Objekt2)* wird damit zu einer relationalen Datenbankabfrage, s.[HIN2].

Hardware-Fehlertoleranz

Bisher betrachteten wir die Toleranz der Assoziativspeicher-Funktionen nur gegenüber fehlerhaften Daten. Gilt diese Toleranz auch für Fehler in der Hardware des Speichers?
Es gibt verschiedene Hinweise aus der Neurologie, daß selbst ein massiver Ausfall von Gehirnneuronen keine großen Auswirkungen auf die Leistungen des Gehirns haben muß.
Analog dazu stellte bei der Modellierung des linearen Assoziativspeichers beispielsweise Kohonen [KOH3,p.114] fest, daß nicht alle Verbindungsgewichte (Speicherelemente) nötig sind, um eine korrekte Funktion zu gewährleisten; eine Relation von 1:40 zwischen der Zahl der

Eingabekomponenten und der Zahl der Verbindungen reicht vollkommen aus. Weder diese noch andere Fehlertoleranzaussagen (s. [WOOD], [BELF]) sind analytisch begründet.
Auch bei dem Modell des nicht-linearen Assoziativspeichers zeigt sich, daß durch Einführung der Schwellwerte große Teile der Hardware ausfallen können, bevor die korrekte Funktion des Assoziativspeichers gestört wird. Dies soll nun im Folgenden genauer quantitativ untersucht werden. Dazu müssen wir uns zunächst ein Modell aller auftretenden Fehler definieren.

Das Fehlermodell
Als Hardwaremodell legen wir uns das in Abbildung 4.3.1 eingeführte Modell zugrunde. Dabei machen wir folgende Annahmen:

- Eingabe -und Ausgabemuster sind binär (binäres Modell, s. 4.1).
- Die Ausgabemuster sind orthogonal kodiert (siehe vorige Abschnitte).
- Die zu speichernden Eingabemuster sind orthogonal kodiert.
- Die Hardwarekomponenten haben nur zwei Fehlzustände: *stuck-at-one* und *stuck-at-zero*.
- Die Hardwareausfälle sind *stochastisch unabhängig*.
- Die Hardwarekomponenten, die ausfallen können, sind die Prozessorelemente (Neuronen) und die Verbindungen.

Aus der zweiten und dritten Annahme folgt direkt für die Gewichte, daß bei $y_i=1$

$$w_{ij} = \quad \Sigma_k \; y_i^k x_j^k \quad = \Sigma_k \; x_j^k \quad = x_j^r$$

die Gewichte der i-ten Spalte nur die Werte des Eingabemusters, also nur 0 oder 1, annehmen können.
Der Ausfall von Neuronen verfälscht zwar direkt die Ausgabe, kann aber in bestimmten Grenzen von nachfolgenden Schichten kompensiert werden. Betrachten wir am Beispiel eines Speichers mit 1000 Eingabeleitungen und 1000 Ausgabeleitungen die Zahl der 10^6 Verbindungen, so bilden die 1000 Prozessorelemente (Neuronen) am Ausgang nur 0.1% der gesamten Hardwareelemente. Bei ungefähr gleicher Komplexität (Zahl der Gatterfunktionen) ist der Einfluß der Verbindungselemente auf die Speicherfunktionen also ungleich stärker und soll deshalb im Folgenden genauer betrachtet werden.
Dazu führen wir eine weitere Verfeinerung des Fehlermodells ein:

- Die stuck-at-one Verbindungen können auf zwei verschiedene Arten modelliert werden:
  aktives Fehlermodell : Die Verbindungen sind immer '1', unabhängig vom Zustand der Eingabe. Beispiel: Binärzähler als Gewichte
  passives Fehlermodell : Die Verbindungen sind nur dann '1', wenn ein Eingabesignal anliegt. Beispiel: Widerstände als Gewichte

Hardware-Fehleranalyse
Mit der Definition

P_0 := P(Verbindung defekt und stuck_at_0)
P_1 := P(Verbindung defekt und stuck_at_1)

wurden in [BR11] Bedingungen hergeleitet, unter denen noch eine korrekte Funktion des Assoziativspeichers erfüllt ist.

Für das *aktive Fehlermodell* ergibt sich die folgende Relation

$$d(\mathbf{x},\mathbf{x}^k) \geq [a - (a+x)(P_0+P_1) + 2nP_1] / (1-(P_0+P_1)) > d(\mathbf{x},\mathbf{x}^r)$$

die eine Beschränkung der Gesamtfehlertoleranz bedeutet: Sind die Datenfehler der Eingabe groß, so dürfen die Hardwareausfälle für eine korrekte Gesamtfunktion nicht zu groß sein und umgekehrt. Dabei gilt allerdings die Nebenbedingung

$$d(\mathbf{x},\mathbf{x}^k) > d_H/2 = a > d(\mathbf{x},\mathbf{x}^r)$$

Betrachten wir die Grenzfälle. Bei maximalen Hardwareausfällen für die reine Erkennung der Klassenprototypen mit $d(\mathbf{x}^k,\mathbf{x}^r) = 2a$ und $d(\mathbf{x}^r,\mathbf{x}^r) = 0$ wird die Relation zu

$$1/4 \geq a/2n \geq P_1 \quad \text{and} \quad 1 > P_0 + P_1, \quad P_0 < 1/2 + P_1\,(n/a - 1)$$

Die stuck_at_1 Verbindungen kompensieren die stuck_at_0 Effekte bis zu einem gewissen Grad. Ziehen wir nur fehlende Verbindungen in Betracht ($P_1 = 0$), so folgt, daß der Speicher bei korrekter Funktionsweise noch das Fehlen bis zur Hälfte aller Verbindungen toleriert.
Betrachten wir umgekehrt den Fall maximaler Datendefekte mit $d(\mathbf{x},\mathbf{x}^r) \rightarrow d/2 = a$. Hier ist $P_0 = (2n/a-1)P_1$; die stuck_at_one Fehler wiegen wesentlich schwerer als die stuck_at_zero Fehler.

Bei dem *passiven Fehlermodell* ergibt sich eine ähnliche Relation wie oben

$$d(\mathbf{x},\mathbf{x}^k) \geq [a - (a+x)(P_0+P_1) + 2xP_1] / (1-(P_0+P_1)) > d(\mathbf{x},\mathbf{x}^r)$$

mit der gleichen Nebenbedingung.
Bei maximalen Hardwareausfällen verändert sich die Relation zu

$$1/2 \geq P_1 \qquad \text{und} \qquad 1/2 > P_0$$

Bei maximalen Datendefekten mit $d(\mathbf{x},\mathbf{x}^r) \rightarrow d/2 = a$ erhalten wir im passiven Fehlermodell $P_0 = P_1$. In diesem Fall dürfen keine zusätzlichen Hardwaredefekte vorliegen; die Effekte vonP_0 und P_1 müssen einander kompensieren.

Diskussion der Ergebnisse

Zweck des Hardwaremodells war es, die inhärenten Fehlertoleranzeigenschaften der nicht-linearen neuronalen Netze am Beispiel des Assoziativspeichers analytisch zu zeigen
Wie wir sahen, ist das Hardwaremodell ziemlich empfindlich gegenüber Kurzschlüssen (stuck_at_1 Fehlern) bei Verbindungsknoten, die technologiebedingt sich als aktive Fehler auswirken. Wird dagegen bei der Hardwareimplementierung eine Technologie verwendet, die passive stuck_at_1 Fehler erzeugen, so ist der Assoziativspeicher, ebenso wie bei allen stuck_at_zero Fehlern, sehr fehlertolerant und toleriert noch einen hohen Ausfall von Komponenten.

Allgemein ergibt sich die Eigenschaft einer "Gesamtfehlertoleranz": Sind die Eingabedaten (Abrufschlüssel des Assoziativspeichers) stark fehlerhaft, so dürfen für ein korrektes Ergebnis nur wenige Hardwareausfälle vorliegen; umgekehrt sind bei starken Hardwareausfällen nur sehr geringe Abweichungen von den gespeichtern Abrufschlüsseln zulässig.

Die Folgerungen daraus sind für den Herstellungsprozeß solcher Schaltungen ambivalent: entweder kann man den Herstellungsprozeß weit weniger aufwendig machen, da mehr Fehler in den Schaltungen erlaubt sind, oder aber man erhält trotz gleicher Technologie eine höhere Ausbeute von

funktionstüchtigen Chips.
Dabei sollte man aber die Voraussetzungen, unter denen die Ergebnisse errechnet wurden, nicht aus den Augen verlieren:

- Das Hardwaremodell ist *sehr einfach.*
  Führt man eine Vielfalt der möglichen Defekte ein, so ergibt sich ein wirklichkeitsgetreueres Modell. Dies ist allerdings erst für eine konkrete Implementierung sinnvoll.

- Die Syndrome stuck_at_one und stuck_at_zero wurden als *stochastisch unabhängig* voneinander angenommen.
  Identifiziert man die Ursachen dieser Syndrome mit Kuzschlüssen (stuck_at_one) und Ausfällen (stuck_at_zero) der Ausgangstreiber der Verbindungen, so mag das durchaus zutreffen. Im Allgemeinen kann man aber nicht davon ausgehen, da eine Vielzahl von sehr unterschiedlichen Defekten gleiche Auswirkungen haben kann. Ohne eine konkrete Implementierung ist es allerdings nicht sinnvoll, konkrete Abhängigkeiten anzunehmen.

- Im Modell wurden *egalisierte Klassenprototypen* benutzt.
  Werden andere Anforderungen an die zu speichernden Muster gestellt, so führt das zu anderen Schwellwerten und damit auch zu anderen Bedingungen für P_0 und P_1.

Insgesamt zeigt sich, daß nichtlineare, neuronale Netze in geeigneter Implementierung ein hohes Maß an potentieller Fehlertoleranz beinhalten, das in herkömmlichen Schaltungen nicht oder nur sehr schwer mit erhöhtem, zusätzlichen Aufwand zur Verfügung gestellt werden kann.

4.3.4 Fehlerdiagnose mit Neuronalen Netzen

Die Fehlerdiagnose mittels neuronaler Netze läßt sich auf die allgemeine Problematik der Diagnose von Zuständen eines Systems zurückführen. Wie wir in Abschnitt 1.5 sahen, gibt es dafür verschiedene Ansätze. Beim Ansatz der Mustererkennung in Abschnitt 1.5.2 wurden alle möglichen, beobachtbaren Syndrome in Klassen eingeteilt, wobei jeder Klasse ein Systemzustand entspricht. Jeder Systemzustand kann dabei im Allgemeinen auf eine Vielfalt von Fehlerzuständen (Defekten) zurückgeführt werde, so daß neben einer Fehlererkennung als Ergebnis der Mustererkennungsoperation (Klassifizierung der Syndrome) nicht zwangsläufig auch eine Fehlerlokalisierung stattfindet. Erst die Zuordnung des wahrscheinlichsten Fehlerzustandes zum beobachteten Syndrom ermöglicht, in einer oder mehreren Stufen eine Fehlerdiagnose durchzuführen.

In Abschnitt 4.1 wurde gezeigt, daß sich neuronale Netze sehr effektiv für Klassifizierungsaufgaben einsetzen lassen. Spezielle, einfach gefertigte Chips mit vorher bestimmten, festen Gewichten sind so als preiswerte, effektive "Diagnosechips" denkbar.

Eine wichtige andere Anwendung besteht darin, die Diagnose aus Beispielen zu lernen. Werden dazu merkmals-erkennende Operationen (s. Abschnitt 4.3.2) benutzt, so läßt sich eine Diagnose sogar auf der Basis von unbekannten Merkmalen erlernen [GRU]. Dabei zeigt sich, daß die Unterschiede in der Diagnose-Charakteristik bei neuronalen Netzen und bei mit den gleichen Beispielen trainierten Menschen relativ gering sind.[GLU].

Auch die klassischen, symbolistischen Ansätze, die mit Hilfe von "tiefem Wissen" [STEE] Diagnose betreiben, haben von ihren Grundmechanismen her starke Ähnlichkeit mit den konnektionistischen Ansätzen. Partitionieren wir das Wissen (Fakten, Klausen) in Wissenseinheiten (Schichten?), so wird das Wissen nur über bestimmte Schnittstellen (Datenwege) abgerufen. Dies kann mit einer Vielzahl von Einzelregeln (Neuronen) erfolgen, die bei Erfüllen aller Bedingungen (AND-Funktion der formalen Neuronen, Abschnitt 4.1) "feuern", oder als paralleler Zugriff auf einen assoziativen Speicher (s. z.B. [SOHN]) verstanden werden.
Modelliert man die Vertrauensfaktoren (Certainty-Faktoren) so, daß sie anstelle einer Post-Kondition in den Prä-Konditionen der "nachfolgenden" Regeln erscheinen, beispielsweise
anstatt

IF a_1 AND a_2 THEN c WITH 0.25
IF b_1 AND b_2 THEN d WITH 0.6
IF c AND d THEN e WITH 0.9

eher

IF a_1 AND a_2 THEN c
IF b_1 AND b_2 THEN d
IF (c WITH 0.25) AND (d WITH 0.6) BIGGER t THEN e

so läßt sich auch das Vertrauensfaktor-gesteuerte Schließen durch die Funktion formaler Neuronen modellieren.

4.4 Simulation und Emulation Neuronaler Netze

Eine der interessantesten Proportionen der neuronalen Netze besteht darin, daß sie oft direkt in einfacher, analoger VLSI-Technik realisiert werden können [VITT2]. Dabei kann man die physikalischen Effekte der Analogtechnik ausnutzen, um nicht nur die Funktionen der formalen Neuronen zu approximieren, sondern sogar die Funktionen ganzer Schichten von Neuronen. Ein Beispiel dafür ist die Entmischung einer linearen Überlagerung von unbekannten Signalquellen mit Hilfe eines Netzwerks [VITT1].

Trotzdem aber werden zur Zeit ausschließlich Simulationen und Emulationen der neuronalen Systeme benutzt, da weder über die grundsätzlichen Mechanismen und Algorithmen, noch über die grundsätzlich geeignete neuronale Architektur Einigkeit besteht. Forschung und Entwicklung gehen rasant voran; es vergeht keine größere Konferenz zu diesen Themen, bei der nicht grundsätzliche, neuartige Ideen präsentiert werden. Aus diesem Grund empfiehlt es sich, sowohl für Grundlagenforschungen als auch für reelle Anwendungen, die in Frage kommenden Algorithmen in einer kontrollierten, mit Hilfsmitteln und Werkzeugen gut ausgestatteten, konventionellen Simulationsumgebung auf einem sequentiellen von-Neumann Computer auszutesten und die optimalen Parameter für das spezifische Problem zu bestimmen.

4.4.1 Hardwarekonfigurationen

Funktioniert der Algorithmus erst zufriedenstellend für das Problem, so ist der weitere Schritt, ihn auf parallele Hardware zu migrieren oder in VLSI zu realisieren, nicht mehr ganz so problematisch. Vielfach reicht es für die Real-time Anwendung sogar aus, den Algorithmus in paralleler Version durch ein Netzwerk von von-Neumann Prozessoren, beispielsweise durch ein Transputernetz, ausführen zu lassen.

Welche Probleme stellen sich dabei ?

In Abschnitt 4.2 wurde bereits erwähnt, daß viele Algorithmen von neuronalen Netzen eine Mischung aus sequentiellen und parallelen Konstrukten darstellen. Dabei läßt sich bezüglich ihrer Wechselwirkungen (Kopplungen) mit Hey [HEY] folgende Einteilung vornehmen:

Wechselwirkungsfreie Parallelarbeit

Betrachten wir zunächst die Algorithmen, bei denen die Daten in Bereiche (*packets*) aufgeteilt unabhängig voneinander bearbeitet werden können.

Diese Aufgabe kann leicht von einem Hauptprozessor (*master* oder *farmer*) erledigt werden, der mehr Aufgabenpakete schnürt, als Prozessoren verfügbar sind. Immer dann, wenn der beauftragte Prozessor (*worker*) sein Ergebnis abliefert, bekommt er eine neue Aufgabe.

Beispiele für dieses *farming* sind die Funktion einer Perzeptron-Schicht und die Algorithmen zur Berechnung von Lichtspiegelungen -und brechungen (*ray tracing*).

Sind weniger Aufgabenpakete als Prozessoren da, so läßt sich der Durchsatz dadurch erhöhen, daß die Ergebnisse als neue Pakete für die nachfolgende Bearbeitung den freien Prozessoren übergeben werden (*pipe-line*).

Als adäquate Hardwarestruktur ist beispielsweise eine sternförmige Kommunikationsstruktur oder ein Bussystem denkbar.

Nachbarschaftsabhängige Parallelarbeit

Bei vielen Algorithmen hängt die Lösung des Problems nicht nur von den eingegebenen Datenpunkten ab, sondern auch von den Lösungen, die für die unmittelbaren Nachbarpunkte gefunden werden.

Bei diesem Problem läßt sich eine gute Prozessorauslastung dadurch erreichen, daß jeweils eine zusammenhängende Datenregion einem einzelnen Prozessor übergeben wird. Die aus der Nachbarschaft benötigten Daten am Rande der Regionen müssen über die Interprozessorkommunikation ausgetauscht werden.

Beispiele dafür sind die parallelen Algorithmen der topologie-erhaltenden Abbildungen aus Abschnitt 4.2.1 oder die numerische Integration partieller Differenzialgleichungen.

Vollvernetzte Parallelarbeit

Werden im Algorithmus die Wechselwirkungen großer Teile des Systems berechnet, wie dies beispielsweise für den autoassoziativen Speicher, die Hopfield-Netze oder das n-Körper Problem der Astrophysik bzw. das Wechselwirkungsproblem bei der Molekülmodellierung durch Atome nötig ist, so lassen sich erfahrungsgemäß gute Resultate erzielen, wenn die Neuronen, Körper oder Atome gleichmäßig den Prozessoren zugeteilt werden und die Prozessoren innerhalb einer Ringschaltung die Eingangsgrößen und Ausgangsgrößen zirkulieren lassen [APPL].

Zur Veranschaulichung sei in Abbildung 4.4.3 die Hard- und Softwarestruktur eines Transputerings zur Modellierung eines biologischen Photosynthesemoleküls [GRUB] wiedergegeben.

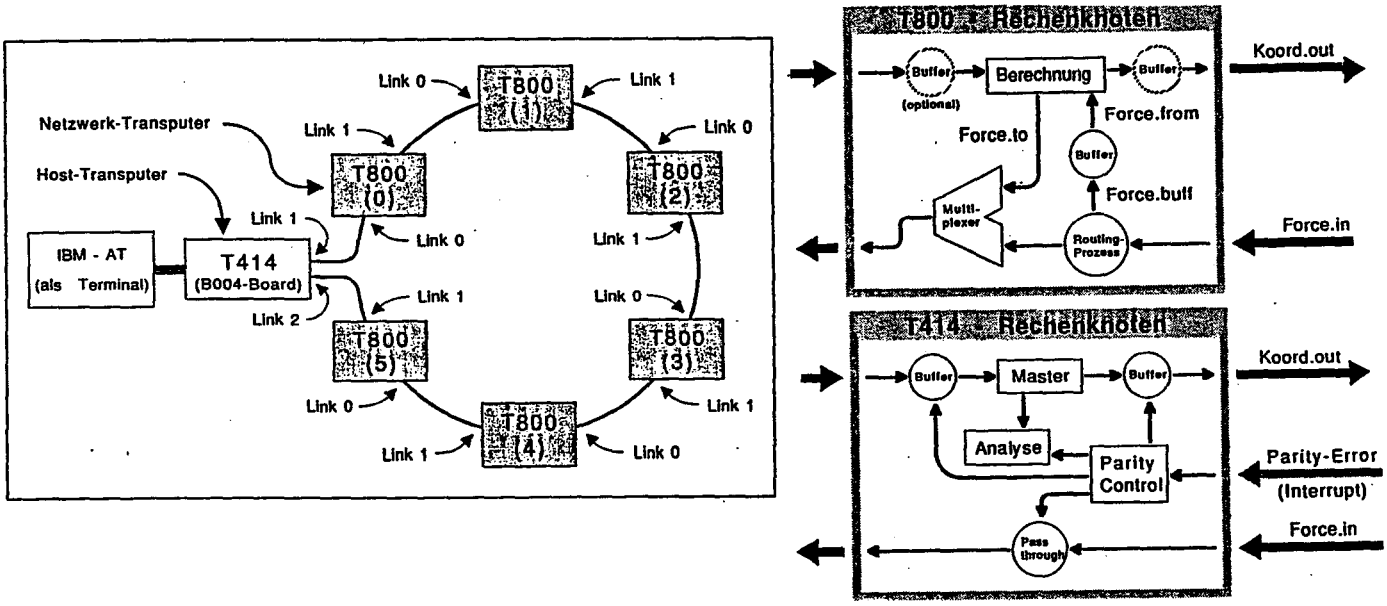


Abb.4.4.3 Hard-und Softwarestruktur für eine Molekülmodellierung

Andere Ansätze sehen kompliziertere, hierarchische Kommunikationsmuster vor [BARN].

Parallelisierung der Simulation

Bei diesen grundsätzlichen Überlegungen wird allerdings nichts darüber ausgesagt, wie ein konkreter Algorithmus mit seinem neuronalen Netz tatsächlich partitioniert werden soll. Betrachten wir beispielsweise den viel verwendeten Back-Propagation Algorithmus. Er besteht aus mehreren Schichten eines Feed-forward Netzes, bei der jede Schicht mit der nachfolgenden vollvernetzt ist.

Im Unterschied zum Multi-Layer Perzeptron (s. Abschnitt 4.1) ist der Lernalgorithmus jeder Schicht durch den jeweils erzielten Fehler (Differenz zwischen gewünschtem und tatsächlichem Ergebnis) in einer stochastischen Approximation gegeben. Er stellt damit eine direkte Erweiterung des Widrow-Hoff Algorithmus auf mehrere Schichten dar.

Es gibt nun verschiedene Möglichkeiten, diesen Algorithmus auf Multiprozessorsysteme zu verteilen. Ohne auf Details einzugehen sind in Abbildung 4.1.1 zwei von drei prinzipiellen Möglichkeiten schematisch gezeichnet, ein dreischichtiges Netzwerk zu parallelisieren.

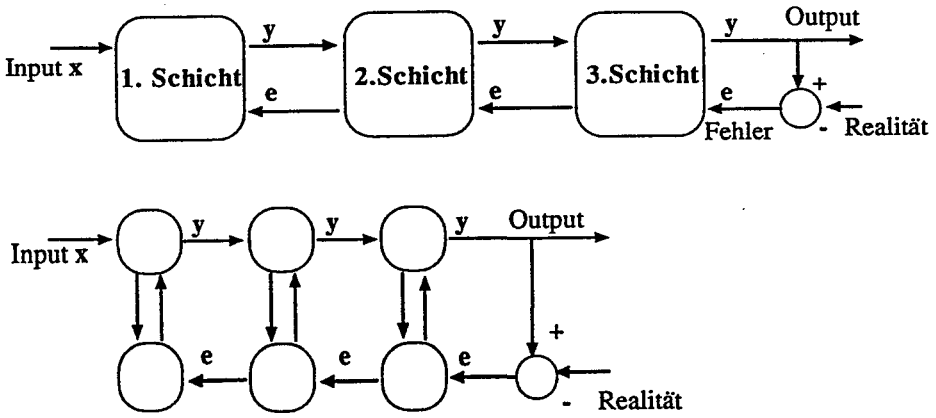


Abb. 4.4.1 Aufteilungen des Back-Propagation Algorithmus

Die erste Aufteilung ordnet jede Schicht einem Prozessor zu, der in einer bidirektionalen Pipeline mit den anderen Prozessoren verbunden ist. Jedes Trainingsmuster, das eingegeben wird, löst eine Aktivität aus, die bis zur Ausgabe geht und als Fehler durch die Schichten bis zur Eingabe zurückgeführt wird. Erst nach Eingabe aller Trainingsmuster werden die Gewichte verändert.

In der zweiten Aufteilung werden die bidirektionalen Aktivitäten getrennt und auf extra Prozessoren verteilt, so daß eine Prozessorstruktur ähnlich einem Hypercube entsteht.

Die dritte Aufteilung partitioniert die Trainingsmuster auf die Prozessoren und führt die Iterationen aller Schichten für diese Teilmenge auf jeweils dem selben Prozessor durch (*farming*).

In ihrer Arbeit [CECI] zeigten L.Ceci, P.Lynn und P.Gardner, daß jede Aufteilung für eine bestimmte Hardwarekonfiguration geeignet ist; dieses Schema ist in Abbildung 4.4.2 gezeigt. Als Kommunikationskosten wurden die eines Intel Hypercubes angenommen.

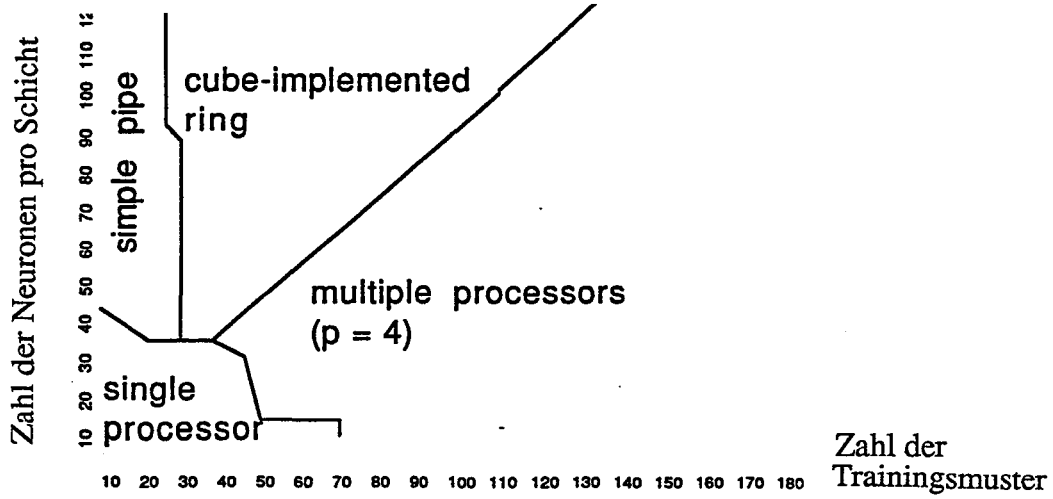


Abb. 4.4.2 Optimale Aufteilung des Algorithmus

4.4.2 Softwarewerkzeuge

Betrachten wir als Beispiel für die Modellierung und Simulation neuronaler Strukturen das folgende Schema des visuellen Kortex.

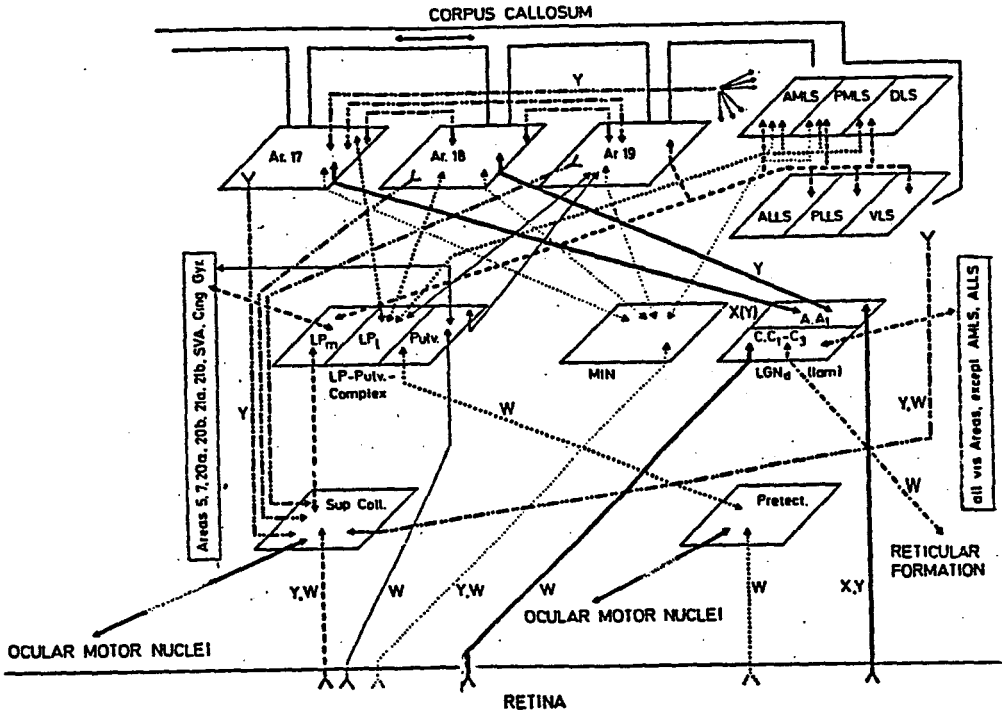


Abb.4.4.4 Schema des visuellen Kortex, aus [SEG]

Wie lassen sich die Eigenschaften dieser vorgeschlagenen Architektur am besten prüfen?

Es ist meist sehr zeitaufwendig und mühsam, ein Konzept einer neuronalen Architektur direkt zur Erprobung in VLSI-Technik zu realisieren oder als Multiprozessorprogramm bis ins Detail auszuprogrammieren. Spielen die Ausführungszeiten gegenüber den qualitativen Merkmalen des gewünschten Algorithmus eine untergeordnete Rolle, so läßt sich eine neuronale Architektur am schnellsten als Simulation auf einem herkömmlichen von Neumann Computer realisieren.

Um den speziellen Bedürfnissen einer solchen Simulation Rechnung zu tragen, gibt es verschiedene Sprachsysteme wie das Lisp-orientierte Spread-3 [DIE], das deklarative Nesila [KOR] oder den bekannten RCS Simulator [GOD], die spezielle Sprachkonstrukte zur Beschreibung der Einheiten und ihrer Verbindungen untereinander bereitstellen. Diese Art der Netzwerkspezifikation ist allerdings sehr mühsam und außerdem (gerade bei größeren Netzen) sehr fehleranfällig. Der Ansatz läßt sich mit dem Assembler-Konzept der Maschinenprogrammierung vergleichen: jedes Detail der neuronalen Maschine muß genau beachtet und ausprogrammiert werden, was bei größeren Programmen (s. obige Abbildung!) selbst bei Anwendung von Modultechnik leicht zu Inkonsistenzen und Fehlern führt. Demgegenüber lassen graphikorientierte, neuronale Simulatoren wie MacBrain™, Cognitron™ usw. zwar meist auch hierarchisch gegliederte, neuronale Netzwerke zu, beinhalten aber neben der Schwierigkeit, die gewünschte Architektur auch mit den fest eingebauten, vorhandenen Grundfunktionen realisieren zu können, auch das Problem der mangelnden Effizienz (Simulationsgeschwindigkeit) bei großen neuronalen Netzen.

Das INES System

Das Interaktive NEtzwerk Simulationssystem (INES), das vom Verfasser entwickelt wurde [BR15], versucht, die Effektivität der dedizierten Programmierung und der Modultechnik mit der Benutzerfreundlichkeit und Fehlervermeidung der graphischen Spezifikation zu verbinden.
Dazu betrachten wir zunächst ein einfaches Simulationsprogramm, das beispielsweise eine Bilderkennung nach Abbildung 4.2.1 durchführen soll. Wie man leicht sehen kann, ist ein solches Programm (falls es gut geschrieben ist) in einzelne Funktionsmodule gegliedert.
Will man nun die einmal geschriebenen Module wiederverwenden (*reusable software*), so bringt man sie günstigerweise in einer besonderen Modulsammlung (*module library*) unter.

Der graphische Editor

Das INES Konzept greift obige Überlegungen auf und ermöglicht die graphische, interaktive Konfiguration bestehender Module im Sinne einer graphischen Programmierung. Die folgende Abbildung zeigt ein Beispiel eines musterverarbeitenden Netzwerks. Die Verbindungen (Pfeile) spezifizieren dabei den Datenstrom, der von den Quellen ('random', 'video') über die funktionalen Einheiten (*Filter*) zu den Senken ("disk","printer","window") fließt. Jedes der umrandeten Bilder (*Ikon*) symbolisiert eine Einheit (*unit*), die selbst wieder aus einem Netz von units (Subnetz) bestehen kann. Auf der untersten Ebene gibt es die Grundeinheiten (*base units*, BasisUnits), die den tatsächlich für die Simulation abzuarbeitenden Maschinencode enthalten und separat in einer geeigneten Programmiersprache geschrieben werden.

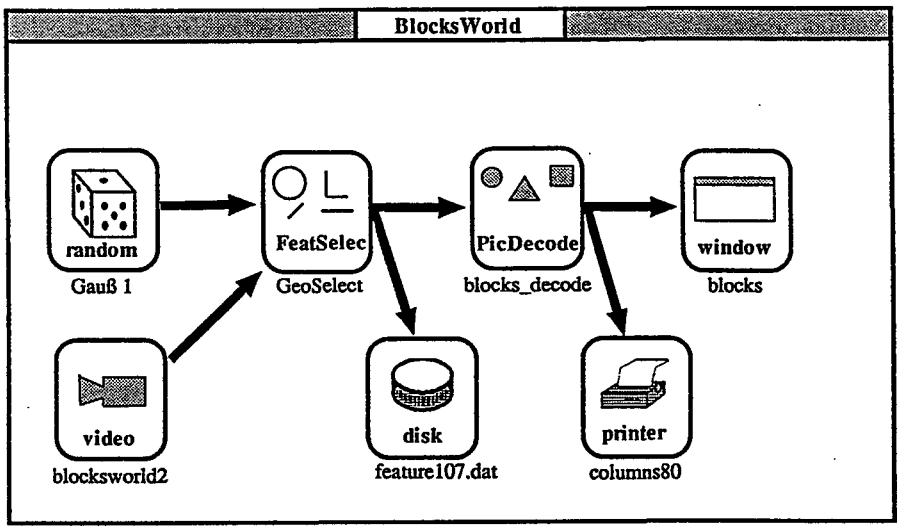


Abb. 4.4.5 Interaktive Spezifikation eines Netzes

Damit ist eine Simulation eines Netzes auf verschiedenen Ebenen möglich. Beispielsweise läßt sich das Netzwerk aus Abbildung 4.4.4, das aus Schichten, Arealen, Funktionsgruppen und Neuronen besteht, sowohl als hierarchisches Subnetz realisieren, bei dem die BasisUnits mit den Neuronen identisch sind, als auch als eine BasisUnit, bei dem alle Verbindungen per Hand programmiert sind.
Welcher Grad zwischen den beiden Extremen gewählt wird, hängt von verschiedenen, benutzertypischen Faktoren ab.
Beispielsweise wird man anfangs nur solche Funktionseinheiten als BasisUnits wählen, die in ihrer

Funktion gut bekannt sind und das unbekannte Zusammenspiel über Monitor- und Protokolliermöglichkeiten (s. obige Abbildung) beobachten. Speziell bei neuronalen Netzen werden zum Debugging spezielle Einheiten benötigt, die den von einer Einheit ausgegebenen Code wieder invertieren, also 'lesbar' machen [KIN].
Ein anderer Aspekt ist die Verteilung auf vorhandene Hardware, wie sie im vorigen Abschnitt diskutiert wurden. Existieren im System spezielle neuronale Chips, beispielsweise als Coprozessoren, oder Netzzugänge zu Multiprozessorsystemen (s. voriges Kapitel 4.4.1), so formuliert man günstigerweise die Netzhierarchie so, daß die Abbildung auf die vorhandenen Hardwareinheiten durch eine passende Spezifikation funktionaler Einheiten unterstützt wird. Jede Hardware-unit entspricht dann einer BasisUnit, die die Treibersoftware zum Ansprechen der Hardware enthält. Entsprechend modifizierte Simulatoren können dann die Verteilung der units auf die Multiprozessorkonfiguration vornehmen.

Die grundsätzlichen Befehle des graphischen Editors wie *move, copy, insert, delete, help, ...* sind orthogonal [SMI] und erlauben Operationen auf einer beliebigen Teilmenge der Netze bzw. Subnetze. Damit läßt sich nur durch Anwendung der Kopier- und Verbindungsoperationen aus einer Kollektion vordefinierter Standardunits (Standardalgorithmen) leicht ein spezielles Netzwerk konstruieren. Dabei wird die Breite jedes Datenpfades mit einer Konstanten durch den Benutzer spezifiziert. Der Editor erlaubt neben dem Abspeichern von Netzen und deren Wiederherstellen auch das Aufsetzen neuer (System) Prozesse. Wird ein Programm aus einer besonders aufgeführten Gruppe ausgewählt, so schreibt der Editor, der auf dem EDGE Konzept [NEW] beruht, die graphischen Strukturen in der Beschreibungssprache GDL ("Graphic Description Language") auf einen Zwischenfile, den das ausführende Programm, beispielsweise ein Simulator oder ein Druckprogramm, seinem Zweck entsprechend auswerten kann.

Der Simulator

Die initialen Funktionen des Simulators bestehen darin, den Zwischenfile des Editors einzulesen, zu interpretieren und alle für die eigentliche Simulation benötigten units zusammenzustellen und entsprechend ihrer Netzstruktur zu initialisieren. Danach wirkt er nur noch als Dispatcher, der gemäß einer vorgegebenen Reihenfolge die einzelnen units durch die Übergabe der Prozessorkontrolle aktiviert. Da selbst für neuronale Chips ('hardware units') eine Initialisierung und Datenübergabe durch eine Treibersoftware nötig ist, kennt der Simulator nur 'software units'. Damit ist es für den Benutzer bei der Netzwerkspezifikation transparent, ob eine unit -Funktion vom Simulator mittels eines speziellen Coprozessors, eines Multiprozessorsystems oder einfach durch sequentielle, lokale Abarbeitung erbracht wird. In der folgenden Abbildung ist dies an dem Beispiel einer pipeline-Funktion (*Filterkette*) gezeigt, die unter anderem einen speziellen Chip benutzt.

Die Implementation dieses Simulatorkonzepts verlangte Designentscheidungen, die wesentlich von der Effizienz einer Implementierung in dem bestehenden Betriebssystem sowie von dem Gesichtspunkt der Portabilität bestimmt wurden.
Eine der wichtigsten Entscheidungen besteht darin, entweder jede unit als eigenständigen (UNIX) Prozeß oder nur als Teil eines einzigen Prozesses zu konzipieren, der alle units enthält. Da die Datenkommunikation zwischen den Einheiten im ersten Fall nur über langsame Nachrichtenverbindungen (s. Kapitel 1.3.3) wie *sockets* oder *pipes* abgewickelt werden können und die Kontrollübergabe zwischen Simulatorprozeß und unit-Prozeß langsame, schwerfällige und fehleranfällige (Deadlocks!) Mechanismen wie globale Semaphoren, dynamische Prioritäten oder

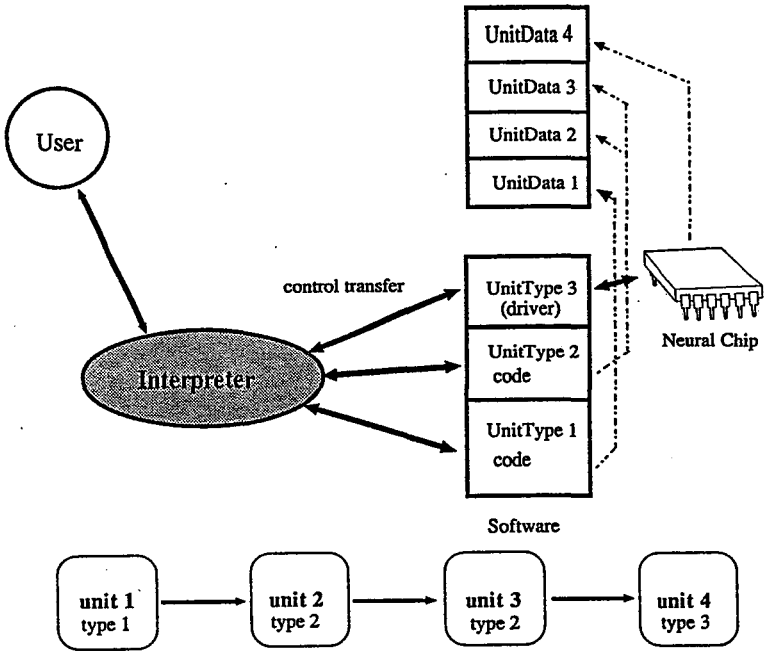


Abb. 4.4.6 Simulation einer inhomogenen Filterkette

file-locking etc. benötigen, wählten wir die Lösung, alle benötigten units wie Leichtgewichtsprozesse (s. Abschnitt 1.3.2) in einem einzigen Systemprozeß zu vereinen.

Dabei ergab sich sich Frage, ob man vom Simulator die Einheiten zur Laufzeit dynamisch nachladen (relozierbarer Code!) oder eine einmalige, statische Linkoperation ausführen lassen sollte. Da unser Compiler bzw. Linker keinen relozierbaren Code erzeugen kann und unser UNIX System durch 'paging' keine merkliche Größenbeschränkung für Prozesse hat, entschieden wir uns für die direkte Linkoperation, die einfacherweise gleich vom Systemlinker ausgeführt werden kann (Portabilität).

Es ergibt sich somit folgendes Zustandsübergangsdiagramm für die Simulation:

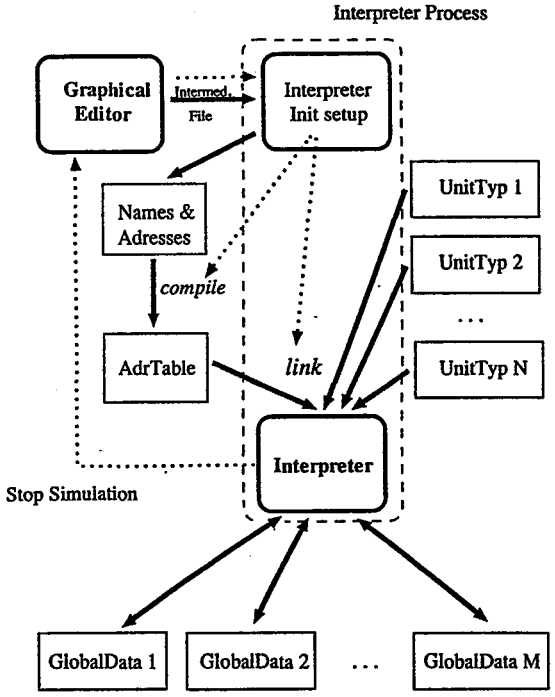


Abb. 4.4.8 Kontroll-, Datenfluß und Erzeugen des Simulators

Da dem Initialisierungsteil des Simulators zwar die Namen, aber nicht die absoluten Adressen der units nach dem Linken bekannt sind, legt er extra eine Adresstabelle an, die mit den unit-Modulen angelinkt und vom Systemlader aufgefüllt wird.

Das nach den Zwischenfile-Spezifikationen (BasisUnits) generierte Simulationsprogramm kann beliebig oft, unabhängig vom Editor, verwendet werden, so daß beim Vorhandensein aller BasisUnits im Simulator beim Übergang vom graphischen Editor kein neuer Simulator generiert werden muß und der Zwischenfile direkt zur Strukturdefinition (Aufbau der internen Datenstrukturen) beim Simulator verwendet werden kann.

Die Strutur einer Basiseinheit

Im graphischen Editor kann der Benutzer neben der Netzwerkeinbindung über die Wahl des Unittyps auch direkt den Algorithmus einer unit angeben. Es ist für Netzwerke mit vielen, gleichartigen Einheiten sinnvoll, in Single-Prozessor Systemen den Algorithmus von den Daten zu trennen und damit auch die Probleme von mehrfachen, gleichen Namen beim Linken zu vermeiden. Die Daten der Ein- und Ausgabe sowie ihre Breite (Puffergröße) müssen von dem unit-Programm-Modul (Algorithmus) nur über Zeiger referiert werden, die über Laufzeitprozeduren vom Simulator zur Verfügung gestellt werden.

Auch die Kontextbereiche jeder unit werden nach dem Einlesen als ASCII-Datei im Speicher im Binärformat abgelegt. Wird die Simulation angehalten, so kann der Benutzer die Kontextdaten ändern. Dazu werden die Daten vom Simulator aus dem Binärformat wieder ins ASCII-Format zurückverwandelt und in einem temporären File gespeichert, das anschließend mit einem normalen Texteditor bearbeitet werden kann. Nach der Rückkehr zum Simulator wird die Datei wieder ins Binärformat zurückverwandelt. Die untenstehende Abbildung 4.4.9 zeigt die verschiedenen Zustände der Kontextdatei einer BasisUnit.

Im Unterschied zu einem Source-Code On-Line Debugger, der ebenfalls eine Änderung der Daten im ASCII-Format bei einem laufenden Programm bereitstellen soll, benötigt die beschriebene Methode keine speziellen Referenzfiles, in denen der jeweilige Compiler Typ und Größe der Daten in einem compilerabhängigen Format festgehalten hat. Der einzige Referenzfile ist der Datenfile (ASCII-Format) selbst, dessen Datenstruktur (Typ, Anzahl und Reihenfolge der Daten) mit der in der BasisUnit verwendeten übereinstimmen muß.

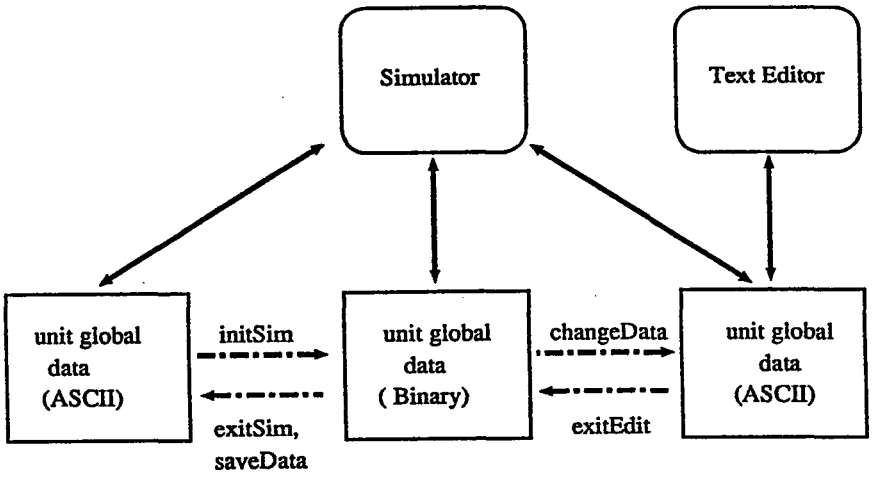


Abb. 4.4.9 Zustände und Übergänge der Kontextdaten einer BasisUnit

Zusammenfassung

Das vorgestellte INES System ermöglicht, flexibel auf die unterschiedlichen Anforderungen an die Simulation neuronaler Algorithmen zu reagieren. Das Software-Design ist dazu mit folgenden Charakteristika versehen worden:

- ♣ Die Netzspezifikation und Programmierung ist interaktiv graphisch möglich
- ♣ Die graphische Spezifikation entspricht dabei einer funktionalen Programmierung
- ♣ Die Wiederverwendung (*reusability*) der Software wird gefördert und durch eine Standardschnittstelle vereinfacht
- ♣ Dabei kann aber auch eine maschinennahe, effiziente Codierung vorgenommen werden.
- ♣ Die Einbindung von benutzerspezifischer, paralleler Hardware ist relativ einfach
- ♣ Die Portabilität wird durch die Verwendung von Standards (C^{++}, UNIX, X-Windows) gefördert

Aber auch die Restriktionen sollen nicht verschwiegen werden:

- Δ Es ist keine feinabgestufte, zeit- oder ereignisgesteuerte Simulation wie beispielsweise mit SIMULA möglich, da der gesamte Ablauf einer BasisUnit als ein Zeitschritt betrachtet wird und externe Ereignisse sich nur in der unit auswirken können, die gerade die Prozessorkontrolle hat.

- Δ Die Verteilung der Prozesse in Multiprozessorsystemen wird mittels der Netzspezifikationen vom Simulator fest vorgenommen. Die günstigste Aufteilung des Algorithmus muß vom Benutzer selbst (durch Aufteilung auf die BasisUnits) vorgenommen werden und wird nicht automatisch generiert, wie dies beispielsweise für das ATTEMPTO 2 System (s.Abschnitt 2.3.5) vorgesehen ist.
  Dies ist bei kleineren Systemen sicher der effektivste Ansatz, um einen gegebenen Algorithmus auf spezielle Hardware (Multi-Transputersysteme etc) verteilen zu können; bei größeren Systemen ist dies aber voraussichtlich zu fehleranfällig und zu arbeitsintensiv.

Im Laufe der Zeit ist deshalb mit dem Wandel der Benutzeranforderungen und der verwendeten Hardware eine Weiterentwicklung des Systems zu erwarten.

Literatur

[AGE] T.Agerwala, Arvind
Data Flow Systems
Computer Vol 14/2, Feb. 1982

[ALM] G.Almasi, A. Gottlieb
Highly Parallel Computing
Benjamin/ Cummings Publ. Corp., Redwood City CA 1989

[AM1] E.Ammann, M.Dal Cin
Efficient Algorithms for Comparison-based Self-Diagnosis
in: M.DalCin, E.Dilger (Eds.), Self-Diagnosis and Fault-Tolerance, pp. 1-18,
ATTEMPTO-Verlag, Tübingen 1981.

[AM2] E.Ammann, R.Brause, M.Dal Cin, E.Dilger, J.Lutz,T.Risse,
ATTEMPTO: A Fault-Tolerant Multiprocessor Working Station, Design and Concepts, Proc. FTCS-13, Milano (1983) 10-13

[AM4] Ammann, E., Brause, R., Dal Cin, M., Dilger, E., Lutz, J., Risse, Th.:
Theoretical Aspects of Test and Diagnosis in ATTEMPTO,
Proceedings of the FTSD'83, CSSR Brünn, 1983, p.84-87

[AM5] E. Ammann
Modelle für die Selbstdiagnose fehlertoleranter Systeme
Dissertation, Fakultät der Physik, Universität Tübingen 1983

[AND] T.Anderson, P.Lee
Fault tolerance, Principles and practice
Prentice Hall, New York 1981

[ANG] B.Ange'niol, G. de la Croix Vaubois, J.-Y. le Texier
Self-Organizing Feature Maps and the Travelling Salesman Problem
Neural Networks Vol 1, pp.289-293, Pergamon Press, New York 1988

[APPL] J.H. Applegate
IEEE Transactions on Computers C-34, 9, pp.882, (1985)

[ARM] J.Armstrong, G.Gray
Fault Diagnosis in a Boolean n-Cube Array of Microprocessors
IEEE Transactions on Computers, Vol C-30, Aug. 1981

[ARN] B.Arnold
Elementare Topologie
Van den Hoek&Ruprecht , Göttingen 1964

[AV] A. Avizienis, L.Chen
On the Implementation of N-Version Programming for Falt-Tolerance During Execution, Proc. COMPSAC 77, pp. 149-155, Nov 1977

[BALD] P. Baldi, K.Hornik
Neural Networks and Principal Component Analysis
Neural Networks, Vol 2, pp.53-58, Pergamon Press 1989

[BALL] D.H.Ballard, Ch. Brown
Computer Vision
Prentice Hall 1982

[BARN] J.Barnes, P.Hut
Nature 324, pp.446-449, 1986

[BAR1] F.Barsi, F.Grandoni, P.Maestrini
A Theory of Diagnosibility of Digital Systems
IEEE Trans. on Comp., Vol C-25, June 1976

[BAR2] F.Barsi
Probabilisitic Syndrom Decoding in Self-Diagnosable Digital Systems
Digital Processes Vol 7, 1981
Georgi Publ. Comp., St.Saphorin, Switzerland

[BELF] L.Belfore, B. Johnson, J.Aylor
The Design of Inherently Fault-Tolerant Systems
in: S.Tewksbury, B.Dickinson, S.Schwartz (Eds)
Concurrent Computations
Plenum Press, New York 1988

[BELL] The UNIX Time-sharing System
The Bell System Technical Journal Vol 57/6 Part2 1978

[BERN] P.Bernstein, V.Hadzilacos, N. Goodman
Concurrency Control and Recovery in Database Systems
Addison-Wesley Publ., 1987

[BEV] W.Bevier
Kit: A Study in Operating System Verifikation
IEEE Transactions on Software Engineering, pp.1382-1396,Vol.15/11, Nov. 1989

[BHU] L.Bhuyan, Q.Yang, D.Agrawal
Performance of Multiprocessor Interconnecting Networks
IEEE Computer, pp. 25-37

[BICH] M.Bichsel, P. Seitz
Minimum Class Entropy: A Maximum Information Approach to Layered Networks
Neural Networks, Vol 2, pp.133-141, Pergamon Press 1989

[BLOM] R.Blomer, C.Raschewa, Rudolf Thurmayr, RoswithaThurmayr
A locally sensitive mapping of multivariate data onto a two-dimensional plane
Medical Data Processing, Taylor & Francis Ltd., London

[BLO] M.Blount
Probabilistic Treatment of Diagnosis in Digital Systems
IEEE Proc. FTCS 7, pp. 72-77, June 1977

[BORG] A.Borg, W.Blau, W.Graetsch, F.Herrmann, W.Oberle
Fault Tolerance under UNIX
ACM Trans. Comp. Syst. Vol 7/1 Febr. 1989, pp.1-24

[BOY] R.S.Boyer, J.S.Moore
A Computational Logic Handbook
Academic, New York 1988

[BR1] R.Brause
Mustererkennung mit stochastischem Lernalgorithmus
Diplomarbeit am Institut für Informationsverarbeitung der Fakultät für Physik der Eberhard-Karls Universität, Tübingen 1978

[BR2] R.Brause, M.Dal Cin
Catastrophic Effects in Pattern Recognition
in: Structural Stability in Physics, Ed. W.Güttinger, H.Eickemeier Springer Verlag Berlin, Heidelberg 1979

[BR3] R.Brause, E.Dilger, Th.Risse
Diagnosing Algorithms and Learning
in: M.DalCin, E.Dilger (Eds.),
Self-Diagnosis and Fault-Tolerance, ATTEMPTO-Verlag, Tübingen 1981.

[BR4] Brause, R., Ammann, E., Dal Cin, M., Dilger, E., Lutz, J., Risse, Th.:
Software-Konzepte des fehlertoleranten Arbeitsplatzrechners ATTEMPTO;
in W.Remmele, H. Schecher (Hrsg.): Microcomputing II,
Teubner Verlag Stuttgart, 1983, p.328-341

[BR4a] R.Brause, E.Ammann, M.DalCin, E.Dilger, J.Lutz, T.Risse
Konzepte des fehlertoleranten Arbeitsplatzrechners ATTEMPTO,
in: E.Maehle, E.Schmitter (Hrs),
Workshop fehlertolerante Mehrprozessorsysteme und Mehrrechnersysteme, Arbeitsberichte des Instituts für mathemat. Maschinen und Datenverarbeitung, Erlangen 1983

[BR5] R.Brause
Selbstdiagnose von Mehrrechnersystemen bei nichtvollständiger Vernetzung
Dissertation an der Fakultät für Physik der Eberhard-Karls Universität Tübingen 1983

[BR6] R.Brause
Design eines fehlertoleranten Rechners
Computer Magazin 11/85 Jahrgang 14, Stuttgart 1985

[BR7] R.Brause
Simulation eines fehlertoleranten Multi-Prozessorsystems unter UNIX™,
Proc. des Workshop der Arbeitsgruppe Simulation von Systemen
ASIM der GI, München 1987

[BR8] R.Brause
Mustererkennung mit verteiltem, assoziativem Speicher
Proc. Workshop Konnektionismus, St.Augustin,
Arbeitspapiere der GMD 1988

[BR9] R.Brause
Prozessoren tauschen Nachrichten über Dual-ported RAMs aus,
VMEbus, Franzis Verlag, München April 1988

[BR10] R.Brause
Fehlertoleranz in intelligenten Benutzerschnittstellen
Informationstechnik it 3/88, pp.219-224, Oldenbourg Verlag 1988

[BR11] R.Brause
Fault Tolerance in Non-Linear Networks
Informatik Fachberichte 188, pp.412-433, Springer Verlag 1988

[BR12] R.Brause
Pattern Recognition and Fault Tolerance in Non-Linear Neural Networks
Proceedings of the Int. Neural Network Society, Boston
Pergamon Press 1988

[BR13] R.Brause
Neural Network Simulation using INES
IEEE Proc. Int. Workshop on tools for AI, Fairfax, USA 1989.

[BR14] R.Brause
Performance and Storage Requirements of Topology-conserving Maps for Robot Manipulator Control, Interner Bericht 5/89 des Fachbereichs Informatik der J.W. Goethe Universität Frankfurt a.M., 1989

[BR15] R.Brause
Performance of Topology-conserving Maps for the Learning of Robot Manipulator Control, Proc. AICSR 89, Elsevier publ. Comp. North-Holland 1989

[CECI] L.Ceci, P.Lynn, P.Gardner
Efficient Distribution of Back-Propagation Models on Parallel Architecture
Report CU-CS-409-88, University of Colorado, September 1988
Abstract in : Proc. Int.Conf. Neural Networks, Boston, Pergamon Press 1988

[CHER] V.S.Cherniavsky, H.Broer
A New Homogenoius Microprogrammable Computer Architecture
in W.Remmele, H. Schecher (Hrsg.): Microcomputing II,
Teubner Verlag Stuttgart, 1983, pp. 159-177

[CIO] P.Ciompi, L.Simoncini
Analysis and Optimal Design of Self-Diagnosable Systems with Repair
IEEE Transactions on Computers C-28/5, May 1979

[CRIS1] F.Cristian, H.Aghili, R.Strong, D.Dolev
Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement
IEEE Proc. FTCS 15, 1985, pp.200-206

[CRIS2] F.Cristian
Agreeing on who is present and who is absent in a synchronous distributed system
IEEE Proc. FTCS 18, 1988, pp.206-211

[DAL1] M.DalCin
Fehlertolerante Systeme
Teubner Verlag, Stuttgart 1979

[DAL2] M.DalCin, E.Dilger (Eds.),
Self-Diagnosis and Fault-Tolerance, ATTEMPTO-Verlag, Tübingen 1981.

[DAL3] M.DalCin, R.Brause, E.Dilger, J.Lutz, T. Risse
ATTEMPTO- A Testable Experimental Multiprocessor System with Fault-Tolerance,
IEEE Computer Architecture Technical Committee, June 1985

[DAL4] M.Dal Cin, R.Brause, E.Dilger, J.Lutz, T. Risse
ATTEMPTO- An experimental fault-tolerant multiprocessor system,
Fachbereich Informatik, Interner Bericht 5/86, Universität Frankfurt 1986

[DAL5] M.Dal Cin, R.Brause, E.Dilger, J.Lutz, T. Risse
ATTEMPTO- An Experimental Fault-Tolerant Multiprocessor System
Microprocessing and Microprogramming 20, pp. 301-308, North Holland 1987

[DAL6] M.Dal Cin
Grundlagen der systemnahen Programmierung
Teubner Verlag 1988

[DAL7] M.Dal Cin
On distributed System-Level Self-Diagnosis
Proc. Fault-Tolerant Comp.Systems, Baden-Baden, pp.186-196
Informatik Fachberichte 214, Springer Verlag 1989

[DAL8] M.Dal Cin, R.Brause, W.Günter
Fehlertoleranz verlangt Mehrfachansatz
Focus 3, Computerwoche 11/8 1989

[DAM] A.Damm
Self-Checking Coverage of Components of a Distributed Real-Time System
Proc. Int. Conf. Fault-Tolerant Comp.Systems, Baden-Baden,
Informatik Fachberichte 214, Springer Verlag 1989

[DE MORI] R.De Mori, C. Suen
New Systems and Architectures For
Automatic Speech Recognition And Synthesis
Springer Verlag 1984

[DEMM] F.Demmelmeier
Fehlertolerante Multimikrorechnersysteme für die Prozeßautomatisierung
Oldenbourg Verlag München Wien 1988

[DEN] J.Denavit, R.S.Hartenberg
A Kinematic Notation for Lower-pair Mechanismen Based on Matrices
Journ. Applied Mech., 1955, Vol 77, pp.215-221

[DIE] J.Diederich, C.Lischka,
Spread-3. Ein Werkzeug zur Simulation konnektionistischer Modelle auf Lisp-Maschinen, KI-Rundbrief Vol 46, pp 75-82, Oldenbourg Verlag 1987

[DIJ] E.W.Dijkstra
Cooperating Sequential Processes
Technological University, Eindhoven, Niederlande 1965

[DIJ2] E.W.Dijkstra
A Discipline of programming
Prentice Hall, Englewood Cliffs, N.J., 1976

[FÄR] G.Färber,
Task-Specific Implementation of Fault-Tolerance in Process Automation,
in [DAL2] pp. 84-102.

[FELD] J.A.Feldman, D.H.Ballard
Computing with connections
University of Rochester, Computer Science Department, TR72, 1980

[FEN] T.Feng
A Survey of Interconnection Networks
IEEE Computer Vol 14/12, Dec. 1981, pp. 12-27

[FLY] M.J.Flynn
Very high speed computing systems
Proc. IEEE 54, Dec 1966, pp. 1901-1909

[FRI] S.G.Frison, J.H.Wensley,
Interactive Consistency and its Impact on the Design of TMR Systems,
Proc. FTCS-12, Santa Monica, (1982) pp.228-234

[FU] Fu, Gonzales, Lee
Robotics: Control, Sensing, Vision and Intelligence
McGraw-Hill 1987

[FUJI] H.Fujiwara, K.Kinoshita
Connection Assignment for Probabilistic Diagnosable Systems
IEEE Transactions on Computers C27, March 1978

[FUK] K.Fukushima
A Neural Network Model for selective Attention in Visual Pattern Recognition
Biological Cybernetics 55, pp 5-15, Springer Verlag 1986

[FUT] G.Futschek
Programmentwicklung und Verifikation
Springer Verlag Wien New York, 1989

[GAL] A.R.Gallant, H.White
There exists a neural network that does not make avoidable mistakes
IEEE Second Int. Conf. on Neural Networks, pp. 657-664, San Diego 1988

[GALL] S.I.Gallant
Connectionist Expert Systems
Comm. ACM, Vol 31/2, Febr. 1988, pp.152-169

[GHE] C. Ghezzi
Concurrency in programming languages: A survey
Parallel Computing vol 2, pp.229-241, Nov. 1985

[GLU] M.Gluck, G.Bower
Evaluating an adaptive network model of human learning
Jounal of memory and language 27, 1988

[GOD] N.Goddard
The Rochester Connectionist Simulator, User Manual and Advanced Programming Manual, Dep. of Computer Sci., University of Rochester, USA, April 1987

[GÖR] W.Görke
Fehlertolerante Rechensysteme
Oldenbourg Verlag, München 1989

[GOS] K.Goser, C. Foelster, U.Rueckert
Intelligent memories in VLSI
Information Sciences 34, p61-82, 1984

[GOTT] A.Gottlieb, C.Krustal
Complexity Results for Permuting Data and other Computations on Parallel Processors, Journal ACM 31, pp.193-209, April 1984

[GOTT2] A.Gottlieb, R.Grishman, C.Krustal, K.McAuliffe, L.Rudolph, M.Snir
The NYU Ultracomputer-Designing an MIMD Shared Memory Parallel Computer
IEEE Transact. on Comp., C-32, No.2î pp-175-189, Febr.1983

[GRAM] T.Grams
Diversitäre Programmierung : Kein Allheilmittel
Informationstechnik it Vol 28/4 (1986), pp. 196-203

[GRI] D.Gries
The science of programming
Springer Verlag Berlin-Heidelberg-New York 1981

[GROS] S. Grossberg
Adaptive pattern classification and universal recoding
Biological Cybernetics, 23, Springer Verlag 1976

[GÜN1] W. Günter
Realisierung des ATTEMPTO Port-Handlers unter UNIX
Bericht für die Deutsche Forschungsgemeinschaft,
Fachbereich Informatik, Universität Frankfurt 1990

[GÜN2] W. Günter
ATTEMPTO: Test, Validierung und Verifikation eines Fehlertoleranzkonzepts für ein Multiprozessorsystem
Dissertation, Universität Erlangen, in Vorbereitung

[GRUB] H.Grubmüller, H.Heller, K.Schulten
Eine Cray für "jedermann"
mc 11/88, Franzis Verlag, München 1988

[GRU] A.Grumbach
Modèles connexionistes du diagnostic
Proc. Journées d'électronique, Ecole Polytechnique Fédérale, Lausanne 1989

[HAK] S.Hakimi, A.Amin
Characterization of Connection Assignment of Diagnosable Systems
IEEE Trans. On Comp., Jan 1974

[HARR] E.Harrison, E.Schmitt
The Structure of Systems/88, a Fault-Tolerant Computer
IBM Systems Journal, Vol 26/3 (1987), pp.292-318

[HEY] A.Hey
Parallel Decomposition of large Scale Simulations in Science and Engineering
Report SHEP 86/87-7, University of Southampton 1987

[HOP] A.L.Hopkins et al.,
FTMP: A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft Control,
Proc. IEEE, Vol 66/10 (1978), pp.1221-1239

[HOR] K.Hornik, M.Stinchcombe, H.White
Multilayer Feedforward Networks are Universal Approximators
Neural Networks, Vol 2, pp.359-366, Pergamon Press 1989

[HILL] D.Hillis
The Connection Machine
MIT Press, Cambridge, Massachusetts, 1985

[HIN1] G.Hinton, Anderson (eds)
Parallel Models of Associative Memory
Lawrence Erlbaum associates, Hillsdale 1981

[HIN2] G.Hinton
Implementing Semantic Networks in Parallel Hardware
in [HIN1]

[JON] A.Jones, P.Schwarz
Experience Using Multiprocessor Systems
ACM Computing Surveys Vol 12, Febr. 1980, pp. 121-165

[JOR] H.Jordan
A Special Purpose Architecture for Finite Element Analysis
Proc. Int.Conf. Parallel Processing, pp.263-266, 1978

[KAM] K.Kammers
Parallelisierung prozeduraler, blockorientierter Programmiersprachen
Diplomarbeit am Fachbereich Informatik
der J.W.-Goethe Universität, Frankfurt a.M. 1989

[KAT] D.Katsuki et al.,
PLURIBUS- an Operational Fault-Tolerant Multiprocessor,
Proc. IEEE, Vol 66/10 (1978) 1146-1159

[KEL] J.Kelly
Current Experiences witgh Fault Tolerant Software Design:
Dependability Through Diverse Formal Specifications?
Proc. Int. Conf. Fault-Tolerant Computing Systems, Baden-Baden ,
Informatik Fachberichte 214, pp.134-149, Springer Verlag 1989

[KIN] J.Kindermann
Inverting Multilayer Perceptrons
Proc. DANIP Workshop on Neural Networks, GMD St.Augustin, April 1989

[KIRR] H.Kirrmann
Industrieller Einsatz fehlertoleranter Rechner
Informationstechnik it 30/3, pp. 186-195, Oldenbourg Verlag München 1988

[KNEIL] H.Kneilmann
Ethernet-Konzentrator für ATTEMPTO
Bericht für die Deutsche Forschungsgemeinschaft,
Fachbereich Informatik, Universität Frankfurt 1990

[KOH1] T. Kohonen
Correlation Matrix Memories
IEEE Transactions on Computers C21 1972

[KOH2] T. Kohonen
Analysis of a simple self-organizing process
Biological Cybernetics Vol. 40, pp. 135-140

[KOH3] T.Kohonen
Self-Organisation and Associative Memory
Springer Verlag Berlin,New York, Tokyo 1984

[KOP1] H.Kopetz, A. Damm, Ch. Koza, M.Mulazzani, W. Schwabl, Ch.Senft, R.Zainlinger
MARS: Ein fehlertolerantes, verteiltes Echtzeitsystem
Informationstechnik it 30/3, Oldenbourg Verlag München 1988

[KOP2] H.Kopetz, A. Damm, Ch. Koza, M.Mulazzani, W. Schwabl, Ch.Senft, R.Zainlinger
MARS: A Fault-Tolerant, distributed Real-Time System
IEEE Micro, Vol /2 Febr.1988, pp. 25-40

[KOR] T.Korb, A.Zell
A declarative Neural Network Description Language
Proc. Euromicro, Kölln 1989, Microprogr. and Microproc., Vol 27/1-5,
North- Holland Publ.

[KUHL] J.Kuhl,S.Reddy
DistributedFault-ToleranceforLargeMultiprocessor
Systems
Proc.7thSymp.onComp.Architect.,LaBaule,France1981

[LAM1] L.Lamport, PM.Melliar-Smith
Synchronizing clocks in the presence of faults
SRI Int., Menlo Park, CA., Feb. 1982

[LAM2] Lamport, L.
Using Time Instead of Timeout for Fault-Tolerant Distributed Systems;
ACM Trans. Program. Lang. Syst. 6.2, 1984, p.254-280

[LEV] M.Levine
Vision in man and machine
McGraw Hill 1985

[LIN1] R.Linsker
From Basic Network Principles to Neural Architecture
Proc. Natl. Acad. Sci., USA, Vol.83, pp7508-7512, 8390-8394, 8779-8783

[LIN2] R.Linsker
Self-Organization in a Perceptual Network
IEEE Computer, pp. 105-117, March 1988

[LIN3] R.Linsker
Towards an Organizing Principle for a Layered Perceptual Network
in Anderson (Ed), Neural Information Processing Systems- Natural and Synthetic,
Amer. Inst. Of Physics, New York 1988

[LIS] Liskov, B.
On Linguistic Support for Distributed Programs;
IEEE Trans. on Software Engineering, VOL SE-8, No. 3, May 1982, p.203- 210

[LJUN] L.Ljung
Analysis of Recursive Stochastic Algorithms
IEEE Transactions on Automatic Control, Vol AC-22/4, August 1977

[LONG] H.C.Longuet-Higgins
Holographic model of temporal recall
Nature 217, 1968, p.104

[LUT1] J.Lutz, R.Brause, M.DalCin, Th.Philipp
Ein Parallelisierungskonzept für ATTEMPTO 2
Interner Bericht 1/89 des Fachbereichs Informatik
der J.W. Goethe Universität Frankfurt a.M., 1989

[LUT2] J.Lutz, R.Brause, M.DalCin, K.Mamers, Th.Philipp
Die Erweiterungen der ATTEMPTO 2 Laufzeitbibliothek
Interner Bericht 2/89 des Fachbereichs Informatik
der J.W. Goethe Universität Frankfurt a.M., 1989

[MAE1] E.Maehle
Self-Test Programs and their Application to Fault-Tolerant Multi-Processor Systems
in: M.DalCin, E.Dilger (Eds.),
Self-Diagnosis and Fault-Tolerance, ATTEMPTO-Verlag, Tübingen 1981.

[MAE2] E.Maehle, H.Joseph
Selbstdiagnose in fehlertoleranten Dirmu Multi-Mikroprozessorkonfigurationen
Proc. Fehlertolerierende Rechnersysteme, Informatik Fachberichte 54,
Springer Verlag Berlin 1982

[MAE3] E.Maehle, K.Moritzen, K.Wirl
A Graph Model for Diagnosis and Reconfiguration and Its application to a Fault-Tolerant Multiprocessor System
IEEE Proc. FTCS-16, Wien 1986, pp. 292-297

[MAH] S.Maheshwari, S.Hakimi
On Models for Diagnosable Systems and Probabilistic Fault Diagnosis
IEEE Transactions on Computers, Vol C25, March 1976

[MAI] G.Maier
Modula-2 Debugger for structured Data
Institut für Automatik und Industrielle Elektronik
ETH Zürich, Switzerland

[MAL] Malek
A Comparison Connection Assignment for Diagnosis of Multiprocessor Systems
Proc. 7th Symp. Comp. Arch., La Baule, France 1981

[MAN] U.Manber
System Diagnosis with Repair
IEEE Trans. on Comp. C29, Oct 1980

[McCUL] W.S.McCulloch, W.H.Pitts,
A Logical Calculus of the Ideas Imminent in Neural Nets
Bulletin of Mathematical Biophysics Vol 5,1943,pp.115-133

[MELL] M.Melliar-Smith, R. Schwartz
Formal Specification and Mechanical Verification of SIFT
IEEE Trans. on Comp., Vol C-31, No. 7 (1982), pp.616-630

[MIN] M.Minsky, Papert
Perceptrons
MIT Press, 1988

[MULL] C.Muller
Modula-Prolog User Manual
Institut für Informatik, ETH Zürich, Juli 1985

[NEW] F.Newbury
EDGE: An Extendible Directed Graph Editor
Internal Report 8/88, Universität Karlsruhe, West Germany

[OJA1] E. Oja, J. Karhunen
On Stochastic Approximation of the Eigenvektors and Eigenvalues of the Expectation of a random Matrix
Report TKK-F- A458 (1981), Helsinki Univ. of Techn., Dept. Techn. Physics

[OJA2] E. Oja
A simplified Neuron Model as Principal Component Analyzer
J. Math. Biology, Vol. 15, 1982, pp267-273

[OJA3] E. Oja
Neural Networks, Principal Components, and Subspaces
Int. Journ. of Neural Systems Vol1/1, pp.61-68, World Scientific, London 1989

[ORNS] S.Ornstein, W.Crowther, M.Kraley, R.Bressler, A.Michel, P.Heart
Pluribus - A reliable multiprocessor
AFIPS Proc. Nat. Comp. Conf., AFIPS press, Arlington Va., 1975

[PAR] Y. Parker
Multi-Microprocessor Systems; Academic Press 1983

[PEA] M.Pearse, R.Shostak, L.Lamport
Reaching Agreement in the Presence of Faults
Comm. of the ACM, Vol27/2, pp.228-234, April 1980

[PERL] M.Perlin, J.-M. Debaud
MatchBox: Fine grained Parallelism at the Match Level
IEEE TAI-89, Proc. Int. Workshop on tools for AI, Fairfax, USA 1989.

[PFAF] E.Pfaffelhuber
Learning and Information Theory
Int. J. Neuroscience, Vol 3, pp.83-88
Gordon and Breach Publ., 1972

[PRE] F.Preparata, G.Metze, R. Chien
On the Connection Assignment Problem of Diagnosable Systems
IEEE Trans. on Electronic Comp. Vol EC-16, 1967

[RIS1] T. Risse, R.Brause, M.DalCin, E.Dilger, J.Lutz
Entwurf und Struktur einer Betriebssystemschicht zur Implementierung von Fehlertoleranz, Tagungsbericht Fehlertolerierende Rechensysteme, Bonn, Informatik- Fachberichte 84, Springer Verlag 1984

[RIS2] Th.Risse
Zur Petri-Netz Modellierung der Interprozessor-Kommunikation in ATTEMPTO
Berichte des Instituts für Informationsverarbeitung 4/85, Universität Tübingen 1985

[RIS3] Risse, Th., Dal Cin, M., Dilger, E.
Zur Verwendung fehlertoleranter Daten-Strukturen im Arbeitsplatz-Rechner ATTEMPTO; Informatik Fachberichte, Springer 1986

[RIS4] Th.Risse
Modelling Interrupt based Interprocessor Communication by Time Petri Nets
Interner Bericht 1/87, Fachbereich Informatik, J.W. Goethe-Universität, Frankfurt a.M. 1987

[RITT1] H.Ritter, K.Schulten
On the Stationary State of Kohonen's Self-Organizing Sensory Mapping
Biolog. Cybn. Vol 54, pp. 99-106, Springer Verlag Berlin 1986

[RITT2] H.Ritter, K.Schulten
Convergence Proporties of Kohonen's Topology conserving Maps
Biological Cybernetics, Vol 60, pp.59 ff, Springer Verlag 1988

[RITT3] H.Ritter, T.Martinetz, K.Schulten
Topology-Conserving Maps for Learning Visuomotor-Coordination
Neural Networks, Vol 2/3, pp. 159-167, Pergamon Press, New York 1989

[ROB] J.Robson
The Pluribus Fault-Tolerant Multiprocessor
IEEE FTCS-9 1979

[SAH] F.Saheban, L. Simoncini, A.Friedman
Concurrent Computation and Diagnosis in Multiprocessor Systems
IEEE Proc. FTCS-11, 1981

[SAHN] R.Sahner, K.Trivedi
Reliability Modeling Using SHARPE
IEEE Trans. Reliability, Vol 36, No.2, pp. 186-193, June 1987

[SAN] T.Sanger
Optimal Unsupervised Learning in a Single-Layer Linear Feedforward Neural Network, Proc. of the Intern. Conf. Neural Networks, Boston 1988, Pergamon press

[SANT] M.Santana, E.Zaluska
A High-level Coordinator in a Distributed Filestore Architecture
Proc. Euromicro 89, pp.423-430
Microprocessing and Microprogramming Vol 27/1-5, North-Holland Publ. 1989

[SEG] Segraves, Rosenquist
The afferent and efferent callosal connections of retinotopic defined areas in cat cortex, J. Neurosci., Vol 8, pp.1090-1107

[SCH] R.F.Schmidt, G.Thews
Einführung in die Physiologie des Menschen
Springer Verlag, Berlin 1976

[SHA] E.Shapiro
A Subset of Concurrent Prolog and Its Interpreter
Technical Report TR003

[SMITH] A.Smith
Cache memories
ACM Comp. Surveys Vol.14, Sept 1982, pp. 473-530

[SMI] D.Smith, C.Irby, R.Kimball, B.Verplanck
Designing the STAR User Interface
Byte, April 1982, pp. 242-282

[SOHN] A.Sohn, J-L. Gaudiot
Multilayer of Ring-structured Feedback Network for Production System Processing
IEEE TAI-89, Proc. Int. Workshop on tools for AI, Fairfax, USA 1989.

[STEE] L.Steels, W. Van de Welde
Learning in second generation expert systems
in: Kowalik (ed), Knowledge based problem solving, Prentice Hall 1985

[SWAN] R.Swan, S.Fuller, D.Siewiorek
Cm* - A modular, multi-microprocessor
AFIPS Proc. Fall Joint Comp. Conf 46,1977, pp.637-644

[TAM] N.Tamura, Y.Kaneda
Implementing Parallel Prolog on a Multi-Processor Machine
IEEE Int. Symp. on Logic Programming, Atlanta City 1984

[THOM] D.Thomas et al.
The analysis of the Performance, Reliability and Life Cycle Cost of Multi-Processor Architecture and their Impact on SENET
Report of the Dep. El. Eng. and Comp. Sc., Carnegie-Mellon University, May 1978

[TIL] A. van Tilborg, L.Wittie
Wave Scheduling: Distributed Allocation of Task Forces in Network Computers
IEEE Proc. 2th Int. Conf. on Distr. Comp. Systems, Paris 1981

[TOH] Y.Tohma,
The SAFE-Project, Tokyo Institute of Technology, private Mitteilung.
Institute for New Generation Computer Technology (ICOT)
Tokyo 108, Japan

[TROB1] R.Trobec, J.Korenini, L.Gyergyek
A Regular WSI-Node Architecture
Microproc. and Microprogramming, Vol 21, pp.75-82
North-Holland 1987

[TROB2] R.Trobec, J.Korenini, L.Gyergyek
Two-dimensional Parallel System Diagnostic
Microproc. and Microprogramming, Vol 25, pp.353-358
North-Holland 1989

[VITT1] E.Vittoz, X.Arreguit
CMOS integration of Herault-Jutten cells for seperation of sources
in: C.Mead, M.Ismail (eds), Analog Implementation of Neural Systems
Kluwer Academic Publ., Norwell 1989

[VITT2] E.Vittoz
Analog VLSI Implementation of Neural Networks
Proc. Journées d'électronique, Ecole Polytechnique Fédérale, Lausanne 1989

[WALD] K.Waldschmidt, D.Tavangarian, G.Roll, M.Strugalla, V.Hochstädter
Ein Assoziativspeicher für schnelle Prozessorsysteme
NTG Fachtagung Mikroelektronik für die Informationstechnik, VDE Verlag 1986

[WEN1] Wensley, J.H.
SIFT - software implemented fault tolerance
AFIPS Proc. Fall Joint Computer Conference, Vol 41, pp. 243-253,
AFIPS Press, Montvale, N.J. 1972

[WEN2] J.H.Wensley, J.H., Lamport, L., Goldberg, J., Green, M.W., Levitt, K.N., Milliar-Smith, P.M., Shostak, R.E., Weinstock, C.B., Berson, D.,
SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control,
Proc. IEEE, Vol.66/10, Oct.(1978), 1240-1255.

[WIL] D. Willshaw
Models of distributed associative memory
Unpublished doctoral dissertion, Edinburgh University (1971)

[WIR] N.Wirth, Programming in Modula-2, Springer Verlag (1982).

[WITT] L.Wittie, A. van Tilborg
MICROS, A Distributed Operating System
for MICRONET, A Reconfigurable Network Computer
IEEE Transact. on Computers Vol C-29/ No 12, Dec. 1980

[YAN] Q.Yang, L. Bhuyan, R. Pavaskar
Performance Analysis of Packet-switched Multiople Bus Multiprocessor Systems
Proc. 8th Real-Time Systems Symp., Dec. 1987, CS Press, Los Alamitos, CA, USA

[YEN] W.Yen, D.Yen, K.-S. Fu
Data Coherence Problems in a Multicache System
IEEE Transact. on Computers C-34, no 1, pp. 56-65, (1985)