Softwaremethoden zur Rückwärtsfehlerbehebung in Hochleistungsparallelrechnern mit verteiltem Speicher

Der Technischen Fakultät der
Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Joachim Hönig

Erlangen 1994

Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung:	8. Juli 1994
Tag der Promotion:	19. September 1994
Dekan:	Prof. Dr. F. Durst
Berichterstatter:	Prof. Dr. M. Dal Cin
	Prof. Dr. K. Echtle

Danksagung

Ich möchte mich bedanken bei Herrn Prof. Dr. Mario Dal Cin für die Betreuung und Begutachtung dieser Arbeit, sowie Herrn Prof. Dr. K. Echtle für die Zweitbegutachtung.

Weiterer Dank gilt meinen Kollegen am Lehrstuhl für ihre Anregungen und ihre Diskussionsbereitschaft.

Erlangen, Juli 1994

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Der Fortschritt in der Rechnerentwicklung, der immer höhere Rechenleistungen bereitstellt, ist nach wie vor ungebrochen. Seitens der Anwendungen entstehen im gleichen Maße immer größere Anforderungen und Aufgaben, die plötzlich im Bereich des Möglichen zu liegen scheinen. Die überwiegende Mehrheit dieser Anwendungen sind Beispiele aus dem technisch-wissenschaftlichen Bereich: aus der Klimaforschung, der Teilchenphysik und der Strömungsmechanik. Viele derartige Aufgaben können mit den bisherigen Superrechnern (Vektorrechner) nicht mehr bewältigt werden. An ihre Stelle treten zunehmend Hochleistungsparallelrechner, in denen eine Vielzahl von einfachen und kostengünstigen Prozessoren zur Lösung einer Aufgabe kooperieren. Je nach Anforderungen eines Problems müssen bis zu mehreren tausend oder sogar zehntausend Einzelrechner zu massiv parallelen Systemen kombiniert werden. Die Skalierbarkeit eines Rechners ist daher ein wichtiges Entwurfskriterium für die Architektur von Parallelrechnern.

Einer Vervielfachung der Prozessorzahl steht allerdings nicht eine im selben Maß steigende Rechenleistung gegenüber. Durch Schwierigkeiten bei der Parallelisierung sowie durch Kommunikation und Synchronisation zwischen den Prozessoren treten mit zunehmender Systemgröße Verluste auf. Außerdem steigt mit der höheren Anzahl der Komponenten, in massiv parallelen Rechnern bis zu mehreren tausend Einzelrechnern, die Wahrscheinlichkeit für das Auftreten von Hardwarefehlern. Ein Parallelrechner, der aus 10000 Einzelrechnern besteht, die jeweils eine MTTF (Mean Time To Failure) von einem Jahr haben, ist durchschnittlich alle 50 Minuten von einem Fehler betroffen. Neben dem Entwurf geeigneter numerischer Verfahren ist daher die Erforschung von geeigneten Fehlertoleranzmechanismen eine wichtige Entwicklungsvoraussetzung für den Entwurf und den Betrieb großer Parallelrechner. Fehlertoleranz ist erforderlich, um zum einen auf solchen Rechnern langandauernde Berechnungen (Rechenzeit bis zu mehreren Wochen) überhaupt erst zu ermöglichen, sowie um zum anderen den Ergebnissen solcher Berechnungen vertrauen zu können.

1.1 Fehlertoleranz in Hochleistungsrechnern

Fehlertoleranz ist immer mit Redundanz und dadurch mit Leistungsverlust verbunden. In Hochleistungsparallelrechnern ist daher zwischen den konträren Zielen Fehlertoleranz und hohe Rechenleistung ein Kompromiß zu finden: Mit der Entscheidung, ein Problem auf einem Hochleistungsrechner zu berechnen, erwartet ein Anwender höchste Leistung. Gleichzeitig bedeutet das Scheitern einer aufwendigen und langwierigen Berechnung einen besonders großen Verlust. Mitunter können durch falsche Ergebnisse große Folgekosten entstehen.

Für den Anwendungsbereich der Hochleistungsparallelrechner sind auf Softwaremethoden basierende Fehlertoleranztechniken besonders geeignet, da Hardwaremethoden wegen des ohnehin bereits sehr hohen Hardwareaufwands nur in einem begrenzten Umfang in Frage kommen. Ein besonders einfaches Verfahren ist die Rückwärtsfehlerbehebung: Bei dieser Technik ist zunächst der Einfluß eines Fehlers zu erkennen und zu lokalisieren (Diagnose). Zur Rückwärtsfehlerbehebung wird der Rechner dann in einen früheren, fehlerfreien Zustand versetzt und der jeweils letzte Programmabschnitt wiederholt. Dazu muß vorher eine garantiert fehlerfreie Zustandskopie (Rücksetz- oder Sicherungspunkt) auf einem sicheren, nichtflüchtigen Speichermedium abgelegt werden. Darüberhinaus muß die Fehlerursache vorübergehender Natur sein, d.h. der Fehler tritt während des Wiederholungsbetriebs nicht mehr auf. Im Falle von dauerhaften Fehlern muß die Ursache vorher durch Rekonfiguration ausgegrenzt oder durch Reparatur beseitigt werden.

Gegenstand dieser Arbeit sind portable, allgemein einsetzbare Softwaretechniken zur Rückwärtsfehlerbehebung: sowohl zur Fehlererkennung, als auch zur Zustandssicherung und zur Fehlerbehebung. Ein wichtiges Kriterium ist dabei der möglichst sparsame Einsatz von Redundanz, sowie, um die Verfahren in massiv parallelen Systemen zu verwenden, ihre Skalierbarkeit. Ein besonderes Augenmerk richtet sich auf die Protokolle, die auf der Basis eines praxisorientierten Modells aufgebaut sein sollten, um ihre effiziente Implementierung zu ermöglichen. Die Protokolle sind für Rechner mit Speicherkommunikation geeignet, wie z.B. für den im Rahmen des Sonderforschungsbereichs 182 „Multiprozessor- und Netzwerkkonfigurationen“ entwickelten MEMSY-Rechner („MEMSY“: **M**odulares, **E**rweiterbares, **M**ultiprozessor **SY**stem).

Angesichts der Schwierigkeiten beim Adaptieren eines sequentiellen Programms auf Parallelrechner, sowie der oftmals nur sehr rudimentär vorhandenen Entwicklungsumgebung, ist der Aufwand zum Erstellen fehlertoleranter Programme mit Rückwärtsfehlerbehebung nur in beschränktem Umfang dem Anwender zumutbar. Einerseits ist es deshalb wünschenswert, eine Form der Rückwärtsfehlerbehebung zu ermöglichen, die für den Anwender unsichtbar (transparent) bleibt und die nicht auf die Mithilfe des Programmierers angewiesen ist. Andererseits wird sich zeigen, daß transparente Verfahren im Vergleich zum nicht transparenten, anwendungsabhängigen Ansatz erheblich aufwendiger sind. Im Sinne der Effizienz ist es sinnvoll, daß der

Programmierer beispielsweise den Umfang und den Zeitpunkt der Zustandssicherung selbst bestimmt, da er eine genaue Kenntnis über das Verhalten seines Verfahrens besitzen sollte. Der Programmierer kann dabei durch eine möglichst einfach zu bedienende Bibliothek unterstützt werden. Auch der Gesichtspunkt der Portabilität spricht für die programmgesteuerte Zustandssicherung, da die transparente Sicherung Eingriffe in das Betriebssystem eines Rechners verlangt. Beide Ansätze wurden für den MEMSY-Rechner erprobt und verglichen.

Außer einer Verbesserung der Fehlertoleranzeigenschaften kann die Rückwärtsfehlerbehebung auch für den fehlerfreien Betrieb eines Rechners Vorteile bringen: Die meisten heutigen Multiprozessoren unterstützen Multiuser-Betrieb nur in Ansätzen, da große Anwendungen oft nur dann mit hoher Effizienz bearbeitet werden können, wenn ihnen sämtliche Resourcen des Systems (z.B. Hauptspeicher) zur Verfügung stehen. Da vorab die Laufzeit eines Programms oft nicht bekannt ist, kann die Nutzung eines Rechners nur schlecht vorausgeplant werden. Funktionen der Rückwärtsfehlerbehebung erweisen sich auch hier als hilfreich. Zur Systemwartung, oder wenn ein wichtigerer Auftrag zu berechnen ist, können langlaufende Programme vorübergehend aus dem System entfernt und später wieder aufgesetzt werden.

Ein großes Problem bei der Betrachtung von Fehlertoleranzmaßnahmen in massiv parallelen Rechnern ist ihre Validierung. Massiv parallele Systeme (mit mehr als tausend Prozessoren) sind nur vereinzelt im Einsatz und arbeiten meist nicht stabil. Die Protokolle konnten daher nur auf vergleichsweise kleinen Rechnern implementiert und erprobt werden. Die Arbeit beschränkt sich darauf, nur in diesem Rahmen eine Bewertung vorzustellen. Es soll aber versucht werden, die Skalierbarkeit der Verfahren, bzw. ihre Eignung für größere Rechner, wenigstens anhand von theoretischen Überlegungen nachzuweisen.

1.2 Inhaltsübersicht

Das nächste Kapitel (Kapitel 2) enthält eine kurze Übersicht über die Grundlagen der Parallelrechner und die Eigenschaften typischer technisch-wissenschaftlicher Anwendungen von Parallelrechnern. Weitere Themen sind eine kurze Beschreibung bekannter Fehlertoleranzverfahren, Techniken zur Fehlererkennung und Probleme der Rückwärtsfehlerbehebung in Mehrprozeßsystemen.

Kapitel 3 behandelt die Anforderungen, die im Betrieb eines Parallelrechners an die Fehlertoleranzeigenschaften zu stellen sind. Insbesondere im Hinblick auf massiv parallele Systeme soll der Stand der Technik dargestellt werden, und es soll aufgezeigt werden, in welchen Bereichen noch Defizite vorhanden sind. Dabei wird sich herausstellen, daß insbesondere im Bereich der Fehlererkennung und auch bei der Zustandssicherung ein hoher Forschungsbedarf besteht.

Zur Fehlererkennung wird deshalb in Kapitel 4 ein neuartiges Softwareverfahren

vorgestellt, die „Kontrollflußselbstüberwachung“. Das Verfahren kann auch in Bereichen mit anderen Anforderungen sinnvoll eingesetzt werden. Die Eigenschaften des Verfahrens wurden auf verschiedenen Rechnersystemen, unter anderem auch einem MEMSY-Knotenrechner, untersucht und die Ergebnisse ausgewertet.

Zum schnellen und effizienten Speichern der Zustandskopien einer Applikation wird das Konzept einer nebenläufigen Zustandssicherung verfolgt. Kapitel 5 beschreibt verschiedene Möglichkeiten. Sowohl der transparente Ansatz, als auch der programmgesteuerte (nicht transparente) Ansatz wurden erprobt und bewertet. Die Frage, wie der Programmierer diese Funktionen in seine Anwendungen einbinden kann, ist ebenfalls Gegenstand dieses Kapitels. Dabei werden Hinweise zur Programmierung und Anwendung eines Rechners aus der Sicht der Fehlertoleranz gegeben.

In Kapitel 6 werden die Protokolle vorgestellt, die Fehlererkennung, Zustandssicherung und Wiederanlauf zu einem Verfahren zur Rückwärtsfehlerbehebung integrieren. Die korrekte Arbeitsweise der Protokolle wird mit theoretischen Überlegungen nachgewiesen. Im Anschluß daran folgt eine Beschreibung der Besonderheiten der Implementierung des Verfahrens im MEMSY-Multiprozessor. Da der am Institut vorhandene MEMSY-Rechner im Vergleich zu den eigentlichen Zielsystemen viel zu klein ist, konnten nur Daten für eine grobe Orientierung gewonnen werden. Davon ausgehend werden Schätzungen für große Systeme erstellt.

Kapitel 2

Grundlagen

Die Konzepte, Eigenschaften und typischen Bestandteile von Multiprozessoren für numerische Applikationen beeinflussen die Rahmenbedingungen und Zielsetzungen, die bei der Umsetzung von Fehlertoleranztechniken in diesem Anwendungsbereich zu beachten sind. Unter diesen Bedingungen wird sich herausstellen, daß unter den Fehlertoleranzverfahren die Technik der Rückwärtsfehlerbehebung für die Anwendung in Hochleistungsparallelrechnern besonders geeignet ist.

Je nach Grad der Autonomie der einzelnen Verarbeitungseinheiten in einem Multiprozessorsystem können sich Probleme ergeben, die in ähnlicher Form aus dem Bereich der verteilten Systeme bekannt sind. Ein hoher Grad an Autonomie der einzelnen verarbeitenden Komponenten erlaubt es einerseits, einzelne Fehler einer Komponente zu verkraften, indem Folgefehler und die Ausbreitung von Fehlern vermieden werden. Andererseits ist ein hoher Grad an Autonomie nur schwer handzuhaben, wie in späteren Kapiteln noch zu sehen ist. In diesem Abschnitt werden zunächst die Eigenschaften und typischen Applikationen von Multiprozessoren kurz vorgestellt.

2.1 Multiprozessoren

Parallelrechner können nach der Flynn'schen Klassifikation in zwei Entwicklungsrichtungen unterteilt werden: zum einen SIMD-Rechner, vertreten durch Feldrechner und Vektorrechner, zum anderen MIMD-Rechner, die aus mehr oder weniger autonomen Prozessormodulen zusammengesetzt sind, die über ein Verbindungsnetzwerk kommunizieren. In „massiv parallelen“ MIMD-Rechnern sollen zukünftig bis zu mehreren tausend oder sogar mehreren zehntausend Verarbeitungseinheiten zu einem Rechnersystem verschaltet werden.

2.1.1 Eigenschaften von Multiprozessorsystemen

Bei MIMD-Rechnern kann zwischen lose gekoppelten und eng gekoppelten Systemen unterschieden werden: Lose gekoppelte MIMD-Rechner sind vorwiegend für Anwendungen mit nur geringer Interaktion zwischen den einzelnen Rechnerknoten (Prozessoren) geeignet, während eng gekoppelte Rechner einen höheren Grad der Interaktion ohne einen deutlichen Leistungsverlust erlauben [HB85]. Als Beispiel für lose gekoppelte Systeme sei z.B. an vernetzte Workstations gedacht, während Multiprozessorsysteme als eng gekoppelt gelten mögen.

Ein anderes mögliches Unterscheidungsmerkmal ist die Art der Interaktion (vgl. [Bel92]). Im „Message-Passing"-Modell kommunizieren die einzelnen Rechnerknoten eines MIMD-Systems mittels Nachrichten. Demgegenüber erlaubt das „Shared Memory"-Modell den Zugriff der Rechnerknoten auf gemeinsamen Speicher, möglicherweise haben dabei sogar alle Prozessoren den gleichen, einheitlichen Adreßraum.

Auf die Rechnerknoten eines Multiprozessors sind Mengen von kooperierenden Prozessen (Applikationen) abgebildet. Mehrere Applikationen können sich einen Multiprozessor teilen, indem sie verschiedene Partitionen verwenden. Mehrere Applikationen oder sogar Mehrbenutzerbetrieb innerhalb der einzelnen Knoten sind üblicherweise nicht vorgesehen, weil davon auszugehen ist, daß eine Applikation bereits alle Resourcen eines Knotens belegt. Ein „Resource-Sharing" ist wegen der Verringerung der Effizienz im allgemeinen nicht akzeptabel.

In manchen Rechnersystemen sind zur Ein-/Ausgabe spezielle Knoten für den Zugriff auf Festplatten und zur Netzanbindung vorgesehen. Gruppen („Cluster") von Prozessoren teilen sich häufig einen Knoten mit Festplattenzugriff.

Das Verbindungsnetzwerk ermöglicht im allgemeinen keine vollständige physikalische Verbindung aller Prozessoren, sondern jeder Rechnerknoten hat nur eine beschränkte Anzahl von direkten Verbindungen zu anderen Knoten. Die Verbindungsstruktur richtet sich nach den Anforderungen, wie z.B. Skalierbarkeit, und nach der Struktur typischer Anwendungen. Verbreitete Topologien sind der Hypercube, sowie zwei- oder dreidimensionale Gitter. Auch hierarchische Bussysteme werden zur Kopplung eingesetzt.

Bei der Speicherkopplung ist der direkte Zugriff aller Prozessoren auf einen gemeinsamen Speicher nur bei kleinen Systemen technisch realisierbar (uniform-memory-access (UMA) Modell, vgl. [Hwa93]). In größeren Systemen ist der gemeinsame Speicher über alle Rechnerknoten verteilt, je nachdem, ob ein Speicherzugriff auf lokalen oder entfernten Speicher erfolgt, schwankt die Zugriffszeit (nonuniform-memory-access (NUMA) Modell). Häufig hat jeder Knoten, wie im Message-Passing Modell, einen privaten Speicher, der für die anderen Knoten nicht zugänglich ist (verteilter Speicher).

Um die Programmierung eines Systems zu vereinfachen, wurden verschiedene Ansätze vorgeschlagen, die die Struktur des Systems für den Anwender verbergen:

virtuell gemeinsamer Speicher („Virtual Shared Memory“) [Li91] oder ein virtuell vollständiges Verbindungsnetz [GHPW90, GS91]. Beim Konzept des virtuell gemeinsamen Speichers wird der lokale Speicher als Cache für einen globalen Adreßraum verwendet. Der Ansatz erfordert durch die anfallenden Kohärenzprotokolle einen erheblichen Implementierungs- und Zeitmehraufwand, darüberhinaus hat er sich in der Praxis noch nicht bewährt. Ein virtuell vollständiges Verbindungsnetz kann mit dem Einsatz spezieller Hardwarebausteine (Hochleistungsrouter) oder speziell optimierter Software umgesetzt werden. Dennoch ist in den meisten Fällen die genaue Kenntnis der Kommunikationstopologie eine wesentliche Voraussetzung, um, unter Berücksichtigung von langsamerer Fernkommunikation und schnellerer Kommunikation in der direkten Nachbarschaft, ein gegebenes Problem auf einem bestimmten Rechner optimal umsetzen zu können. Selbst mit dieser Optimierung bildet die Kommunikationsbandbreite häufig einen Engpaß. Ein anderer Ansatz versucht deshalb, mittels „Multi-Threading“ Wartezeiten in der Kommunikation durch schnelle Prozeßwechsel zu maskieren. Insbesondere im Zusammenhang mit Transputer-Architekturen [Inm88] wird dieser Ansatz immer wieder erwähnt.

Das Betriebssystem von Parallelrechnern ist in den meisten Fällen eher minimalistisch ausgelegt. Die einzelnen Rechnerknoten sind nur mit einer Laufzeitumgebung zum Starten von Programmen, zum Aufbau von Verbindungen und zur Kommunikation ausgestattet. Zur Vorbereitung der Aufgaben, also für die Interaktion mit dem Anwender, zur Programmentwicklung, zur Verwaltung der Rechenaufträge, sowie zur Ein- und Ausgabe dient im allgemeinen ein handelsübliches Rechnersystem („Host“). Der aktuelle Trend führt für Parallelrechner aber zunehmend in Richtung der mächtigeren Betriebssysteme, um ein gewisses Mindestmaß an Programmierhilfsmitteln und Komfort bereitzustellen. Beispiele für parallele Hochleistungsrechner sind Rechnersysteme wie nCUBE [nCU90], Intel Paragon [Int91], CM-5 [Thi91], KSR-1 [Bur92], Fujitsu VPP500 [Fuj92], Convex [Con93] und MEMSY.

2.1.2 Der MEMSY-Multiprozessor

Das Projekt MEMSY dient zu Untersuchungen auf dem Gebiet der massiv parallelen Hochleistungsrechner für vorwiegend numerische Anwendungen. Die im Rahmen dieser Arbeit vorgestellten Protokolle wurden für den MEMSY-Multiprozessor implementiert und dort erprobt.

2.1.2.1 Hardware

Ein MEMSY-Multiprozessor besteht aus einer theoretisch unbegrenzten Anzahl von unabhängigen Rechnerknoten [HDG+93, DHHP94]. Bei den Rechnerknoten handelt es sich um MVME188-Systeme von Motorola. Ein Knoten ist für sich gesehen bereits selbst ein Multiprozessor, denn jeder Knoten enthält bis zu vier Prozessoren,

die auf den Hauptspeicher des Knotens direkt zugreifen können. Bei den Prozessoren handelt es sich um Mikroprozessoren des Typs MC88100, die als wesentliches Architekturmerkmal getrennte Pfade und Caches für Daten und Instruktionen besitzen (Harvard-Architektur). Jeder Knoten ist mit einer lokalen Festplatte für den Systemstart und als Hintergrundspeicher ausgestattet.

Zur Kommunikation werden verteilte gemeinsame Speicher und lokale Netze (FDDI und Ethernet) verwendet. Die Hardware für den gemeinsamen Speicher wurde speziell entwickelt [Hil92]:

- Der Kommunikationsspeicherbaustein enthält den eigentlichen Speicher sowie zwei Ports, die mit den entsprechenden Ports eines Prozessorknotens verbunden werden können. Zwischen Prozessor und Kommunikationsspeicher können Koppelmodule geschaltet werden, um verschiedene Topologien zu ermöglichen.

- Das Koppelmodul hat vier Anschlüsse für Prozessoren und ebenfalls vier Anschlüsse für Kommunikationsspeicher. Durch eine Steuerlogik kann dynamisch jeder Prozessoranschluß über vier interne Schaltelemente mit jedem Speicheranschluß verbunden werden.

Abbildung 2.1 zeigt die Grundkonfiguration, einen Torus mit 16 Rechnerknoten. Jeder Knoten besteht aus dem beschriebenen Prozessormodul und einem Kommunikationspeicher, der über je zwei verschiedene Koppelmoduln bzw. Pfade erreicht werden kann. Die Verbindungsstruktur ist skalierbar: Mit einer Erweiterung des Systems um zusätzliche Rechnerknoten wächst das Kommunikationsspeichernetzwerk ebenfalls.

2.1.2.2 Programmiermodell und Betriebssystem

Der Programmierer hat die Auswahl zwischen einer Vielzahl von Mechanismen zur Kommunikation und Synchronisation:

- **Shared-Memory.** Im Kommunikationsspeicher können gemeinsame Speicherbereiche angelegt werden. Diese Speicherbereiche können von Prozessen in direkt benachbarten Knoten in den eigenen Adreßraum abgebildet werden.

- **Nachrichten.** Nachrichten können an beliebige Prozesse einer Applikation auf beliebigen Prozessoren verschickt werden. Das Routing wird vom Betriebsystem übernommen. Unterschieden wird zwischen kurzen Nachrichten („Messages“) und langen Nachrichten („Transports“). Als Übertragungsmedium wird der Kommunikationsspeicher verwendet.

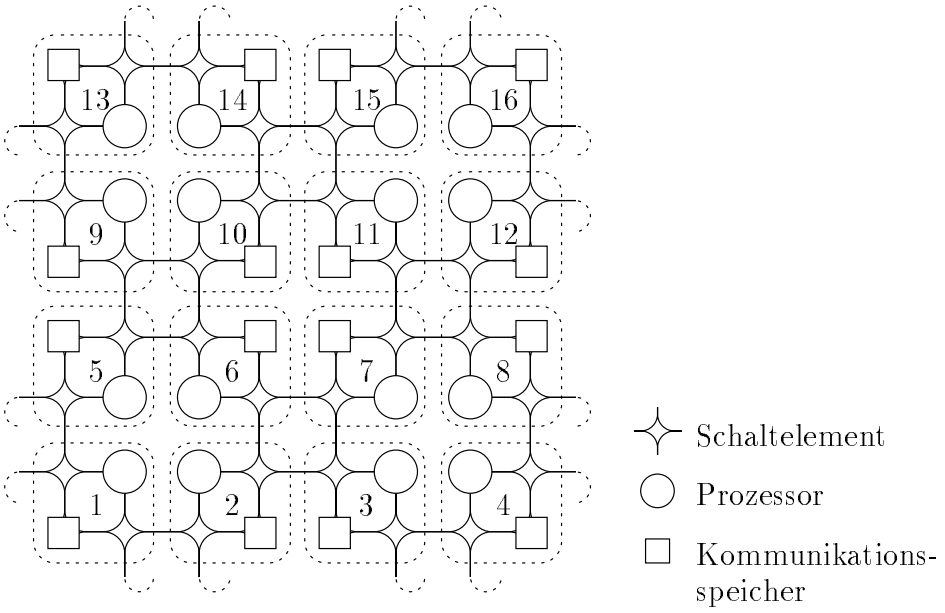

Abbildung 2.1: MEMSY-Kommunikationsspeichernetzwerk

- **Semaphoren und Spinlocks.** Jeder Knoten kann Semaphoren anlegen, auf die global zugegriffen werden kann. Zum Absichern kurzer Abschnitte sind Spinlocks geeignet, die allerdings ein „busy-wait“ ausführen. Spinlocks werden im Kommunikationsspeicher angelegt, können also nur lokal zwischen benachbarten Prozessorknoten zur Synchronisation verwendet werden.

- **Signale.** Ein Prozeß kann einem anderen Prozeß derselben Applikation auch auf anderen Prozessoren ein Signal zustellen. Die notwendige Information wird mittels Nachrichten übertragen.

Als strukturierendes Merkmal wurde im MEMSY-Programmiermodell der Begriff der Applikation eingeführt, der alle Prozesse eines Anwenderprogramms umfaßt. Nachrichten können nur zwischen Prozessen derselben Applikation ausgetauscht werden. Auch gemeinsame Speicherbereiche (Shared-Memory) sind einer Applikation zugeordnet. Nur von Prozessen dieser Applikation darf auf das Shared-Memory-Segment zugegriffen werden.

Um die einzelnen Rechnerknoten optimal auszulasten, kann der Benutzer innerhalb eines Knotens eine der Prozessorenzahl entsprechende Anzahl an gleichzeitig lauffähigen Arbeitsprozessen erzeugen. Spezielle Scheduling-Strategien versuchen innerhalb eines Rechnerknotens, Prozessen der gleichen Applikation gleichzeitig einen Prozessor zuzuteilen.

Im Unterschied zu den sonst üblichen Betriebssystemen mit minimalem Funktionsumfang basiert das Betriebssystem MEMSOS auf einer Adaption des kompletten UNIX System V Release 3. Diese Basis wurde um Funktionen zur Ansteuerung der zusätzlichen Hardware erweitert, wie z.B. des Kommunikationsspeichers [LSTT94].

Eine Besonderheit des MEMSY-Multiprozessors ist der Mehrbenutzerbetrieb. Die Knoten haben unabhängige Betriebssystemkerne und können im Fehlerfall einzeln gestartet werden. Ein MEMSY-Multiprozessor kann einerseits als ein lose gekoppeltes Netz aus unabhängigen Workstations betrachtet werden, andererseits, durch den gemeinsamen Speicher, sind auch Eigenschaften eng gekoppelter Systeme gegeben.

Auf jedem Rechnerknoten läuft ein Hintergrundprozeß zum Starten von Applikationen. Ausführbare Programme werden auf einem von allen Knoten zugänglichen Dateisystem abgelegt. Beim Start erhalten die Hintergrundprozesse aller beteiligten Knoten den Auftrag, das Programm vom globalen Dateisystem zu laden und zu starten. Nach dem Start kann ein Applikationsprozeß seine Position innerhalb der Topologie des Systems durch spezielle Systemaufrufe erkennen.

2.1.3 Typische Applikationsprogramme

Hochleistungsparallelrechner werden für rechenintensive, numerische Applikationsprogramme aus dem technisch-wissenschaftlichen Bereich eingesetzt. Diese Applikationen können durch folgende Merkmale charakterisiert werden:

- sehr hoher Rechenzeitbedarf
- sehr hoher Speicherbedarf
- keine Echtzeitanforderungen
- nicht sicherheitskritisch

Die Laufzeit solcher Programme liegt im Bereich von einigen Stunden bis zu mehreren Wochen. Typische Applikationen sind als oft mehrmals geschachtelte Iterationsschleifen strukturiert. Nach wie vor basieren diese Programme auf Portierungen von sequentiellen Fortran-Programmen, die je nach Erfahrung des Programmierers mehr oder weniger effizient parallelisiert werden. Üblicherweise bearbeiten alle Prozessoren das gleiche Programm für verschiedene Untermengen des Datenbereichs.

Parallele Applikationen kann man, je nach Kommunikationsverhalten und Art der Ablaufsteuerung, grob in zwei Kategorien einteilen: „lose“ und „eng“ kooperierende Applikationen. Lose kooperierende Applikationen zeichnen sich im allgemeinen durch einen hierarchischen Aufbau aus. Ein besonderer Prozeß vergibt Aufträge von etwa gleicher Komplexität an eine Menge von „Worker“-Prozessen, die diese Aufträge unabhängig voneinander bearbeiten und die Ergebnisse schließlich an den

Auftraggeber zurücksenden. Diese Vorgehensweise ist auch unter dem Begriff „Job Farming“ geläufig. Dazu ist es notwendig, daß das Problem in weitgehend unabhängige Aufträge aufgeteilt werden kann. Da sich Parallelrechner durch enge Kopplung und hohe Kommunikationsleistung auszeichnen, werden solche Verfahren nur selten auf Parallelrechner portiert. Lose gekoppelte verteilte Systeme, beispielsweise vernetzte Workstations, sind kostengünstiger, universeller einzusetzen und daher besser für solche Aufgaben geeignet. Ein Beispiel dieser Art von Applikationen sind parallele Raytracing-Verfahren [Fel88].

Wenn unabhängige Arbeitsbereiche nicht gegeben sind und ein hohes Kommunikationsaufkommen besteht, entsteht eine eng kooperierende Applikation. Diese Art der Applikationen ist dadurch gekennzeichnet, daß eine Menge von Prozessen parallel an einer Aufgabe arbeitet. Meistens wird das Problem zu Beginn der Berechnung statisch in etwa gleich aufwendige Teilprobleme aufgeteilt. Nach jedem Berechnungsschritt müssen Zwischenergebnisse ausgetauscht werden. Oft ist ein globaler Datenaustausch zur Kontrolle des Rechenfortschritts nötig. Der ständige Datenaustausch erzwingt globale Kommunikation und oft sogar eine Synchronisation aller Teilberechnungen. Ein Beispiel sind Verfahren zur parallelen Lösung partieller Differentialgleichungen. Mehrgitterverfahren [Hac85] zeichnen sich gegenüber herkömmlichen Verfahren durch erheblich schnellere Konvergenz aus, andererseits ist aber der Programmier- und Rechenaufwand einer Iteration viel höher. Die meisten Klimamodelle und Berechnungen aus dem Bereich der Strömungsmechanik basieren auf solchen Verfahren.

2.2 Fehlertoleranzverfahren

Fehlerhaftes Verhalten eines Rechnersystems kann durch mechanische und elektrische Einflüsse, chemische Veränderungen und Programmierfehler ausgelöst werden. Fehlertolerante Systeme sind in der Lage, trotz des Vorhandenseins von Fehlern die spezifizierte Leistung zu erbringen. In diesem Rahmen seien nur kurz die verschiedenen Verfahren rekapituliert, die in fehlertoleranten Rechnersystemen eingesetzt werden. Eine ausführlichere Behandlung ist z.B. in [Dal79, AL81, Gör89, Ech90] zu finden. Eine kurze, mehrsprachige Übersicht über die Terminologie aus dem Bereich der Fehlertoleranz findet sich in [Lap92].

Toleranz gegenüber Fehlern kann nur durch das Einsetzen von Redundanz erreicht werden, d.h. das korrekte Arbeiten eines Systems muß mittels zusätzlicher Kontrollrechnungen oder Hardwarebauteile überprüft werden. Dabei können statische und dynamische Formen der Redundanz unterschieden werden; Mischformen werden als Hybridredundanz bezeichnet.

Statisch redundante Systeme verwenden Komponenten oder Mittel, die einzig und allein Fehlertoleranzzwecken dienen. Ein Beispiel dafür sind **fehlermaskierende** Systeme wie die TMR-Schaltung (Triple Modular Redundancy) [BGH87], in der

über die Rechenergebnisse dreier gleichartiger Systeme votiert wird, so daß sich ein Fehler nach außen hin nicht manifestiert. Maskierende Systeme werden bevorzugt bei sicherheitskritischen Anforderungen und Echtzeitsystemen eingesetzt, insbesondere weil im Fehlerfall unterbrechungsfreier Betrieb möglich ist. Wegen des hohen Hardwareaufwands durch die Verdreifachung der Komponenten und wegen der aufwendigen Synchronisation der Komponenten sind solche Systeme vergleichsweise teuer. Fehlererkennende bzw. fehlerkorrigierende Codes (ECC) sind aber z.B. zum Erkennen bzw. zum Schutz vor Speicherfehlern weit verbreitet. Statische Redundanz gegenüber Software- und Spezifikationsfehlern kann auch mittels Software realisiert werden, z.B. in der Form der N-Versionen Programmierung [Avi85]. Für die Lösung eines Problems werden mehrere verschiedene Programme von verschiedenen Entwicklerteams eingesetzt.

Zu den **fehlertolerierenden** Systemen gehören vorwiegend dynamisch redundante Systeme. Solche Systeme setzen Redundanz erst im Ausnahmebetrieb nach einem Fehler ein. Zunächst gilt es Fehler zu erkennen, z.B. durch Überprüfungen (Tests) oder durch Überwachung z.B. der Speicherzugriffe. Anschließend sind die Fehler zu lokalisieren und zu beheben. Typischerweise erbringen fehlertolerierende Systeme beim Auftreten eines Fehlers zunächst für eine gewisse Adaptionszeit keine oder nur verminderte Leistung. Zur Fehlerbehebung können wiederum zwei Techniken unterschieden werden: die Rückwärtsfehlerbehebung und die Vorwärtsfehlerbehebung. Bei beiden Verfahren steht während der fehlerfreien Abarbeitung die volle Rechenleistung des Systems zur Verfügung. Im Fehlerfall wird eine relativ zur Gesamtrechenzeit kleine Verzögerung in Kauf genommen (Zeitredundanz), um das System wieder in einen fehlerfreien Zustand zu versetzen.

Zur Vorwärtsfehlerbehebung sind genaue Kenntnisse über die erlaubten Zustände nötig, die das System einnehmen darf. Vorwärtsfehlerbehebung greift nicht auf Zustandsinformation der Vergangenheit zurück, sondern bringt das System nach einem Fehler in einen Zustand, der aufgrund der Spezifikation möglicherweise in der Zukunft aufgetreten wäre [BEG86]. Ein Beispiel für die Vorwärtsfehlerbehebung wäre innerhalb eines Rechnersystems das Beseitigen einer Inkonsistenz im Dateisystem: Durch Entfernen der inkonsistenten Dateien und Verwaltungsinformationen wird das System wieder in einen zulässigen Zustand gebracht, wobei möglicherweise aber ein Verlust von Information oder Rechenfortschritt auftreten kann.

Zur Rückwärtsfehlerbehebung werden in regelmäßigen Abständen Zustandskopien der Prozesse (bestehend aus Datensegmenten, den Prozessorregistern, Prozeßkontrollblock) auf einen nichtflüchtigen, sicheren Speicher („stabiler Speicher“, [Lam88]) ausgelagert. Im Fehlerfall werden betroffene Prozesse auf diese Sicherungspunkte (Rücksetzpunkte) zurückgesetzt. Ein Bestandteil des Verfahrens ist die Annahme, daß im Wiederholungsbetrieb, also nach dem Zurücksetzen, der Fehler nicht mehr auftritt, d.h. daß die Fehlerursache von vorübergehender Natur war (temporärer Fehler). Falls keine Zustandskopien angelegt werden, ist die einfachste Art der Rückwärtsfehlerbehebung der Neustart des Programms von Anfang an, was aber

den Verlust des gesamten Rechenfortschritts bedeutet. Ein ähnlicher Ansatz, speziell für die Toleranz gegenüber Softwarefehlern, ist das Rücksetzblock-Verfahren [Ran79]. Nach dem Fehlschlagen eines Akzeptanztests [And79] am Ende eines Programmabschnitts wird derselbe Abschnitt, vorzugsweise nach einem anderen Verfahren, wiederholt.

Im Falle von dauerhaften Hardwarefehlern (permanente Fehler) müssen dynamische Redundanztechniken wie „Fail-Soft“ (sanfter Leistungsabfall) und „Standby-Sparing“ (Einsatz von Reservekomponenten) verwendet werden. Bei Fail-Soft werden im Fehlerfall defekte Komponenten ausgegrenzt und die Anwendung so rekonfiguriert, daß nur noch die intakten Komponenten verwendet werden. Bei Standby-Sparing werden von vornherein Komponenten reserviert, die im Fehlerfall die fehlerhaften Komponenten ersetzen. Die Methode kann als ein Hybridverfahren statischer und dynamischer Techniken gelten. Auch in maskierenden Systemen kann Rückwärtsfehlerbehebung eingesetzt werden [EN91].

Mit dem im vorherigen Abschnitt genannten Profil der Anwendungen scheiden für Hochleistung-Parallelrechner, die wegen des hohen Leistungsbedarfs in ihrem Hardwareaufbau auch ohne Fehlertoleranzmaßnahmen bereits die Grenzen des technisch und finanziell Machbaren erreichen, statische Redundanztechniken aus, zumal ein definiertes Echtzeitverhalten nicht erwartet wird. Unter den gegebenen Bedingungen kommt in einem Multiprozessor für numerische Anwendungen nur die Rückwärtsfehlerbehebung als Fehlertoleranztechnik in Betracht. Die Vorwärtsfehlerbehebung ist in den meisten Fällen nicht möglich, da, in der Natur der Probleme begründet, im voraus keine genaue Kenntnis über mögliche Ergebnisse besteht. Aus der Spezifikation bestimmter Algorithmen können jedoch immerhin beispielsweise Konvergenzkriterien abgeleitet werden, die im Verlauf der Berechnung immer gültig sein müssen. Die Überprüfung solcher Kriterien kann zur Fehlererkennung verwendet werden [Böh94].

2.3 Fehler und Fehlererkennung

In fehlertolerierenden Systemen müssen Fehler zunächst erkannt werden, bevor Maßnahmen zu ihrer Behebung begonnen werden können. Dabei stellt sich die Frage, welches Verhalten des Systems als fehlerhaft betrachtet werden soll. Ein Fehler sei durch die folgende Definition bestimmt:

> *Ein fehlerfreies Anwendungsprogramm ist genau dann von einem Fehler betroffen, wenn es mit einem falschen Ergebnis oder überhaupt nicht terminiert.*

Fehler nach dieser Definition können durch Fehlverhalten der Hardware bewirkt werden, aber auch Softwarefehler des Betriebssystems sind eingeschlossen. Verfahrens-

und Programmierfehler des Anwendungsprogramms sind ausgenommen, denn die Rückwärtsfehlerbehebung erhebt nicht den Anspruch, diese Fehler beheben zu können. Wenn sich eine Systemkomponente wegen eines physikalischen Einflusses nicht nach der Spezifikation verhält, dieses Fehlverhalten aber keine Auswirkung auf den Rechenverlauf hat, so ist dies kein Fehler im Sinne der Definition.

Die durch obiges Fehlermodell bestimmten Fehler gilt es mittels Überprüfungen (Tests) zu erkennen. Die Güte eines Tests wird durch die Begriffe Fehlerüberdeckung und Latenz charakterisiert. Die Fehlerüberdeckung ist ein Maß dafür, welcher Anteil der Fehler aus dem Fehlermodell erkannt wird und dient als Erwartungswert der Fehlererkennungswahrscheinlichkeit, wenn alle Fehler innerhalb des Modells gleich wahrscheinlich sind. Die Latenz gibt die durchschnittliche Verzögerung zwischen dem Auftreten eines Fehlers und seiner Entdeckung an. In realen Systemen ist die Fehlerüberdeckung kleiner als 100%, und die Latenz ist größer als Null. Auch der zusätzliche Aufwand, der durch einen Test entsteht (z.B. redundante Berechnungen, zusätzliche Hardware) ist eine Eigenschaft, die berücksichtigt werden muß.

Besondere Bedeutung hat die Überdeckung von Fehlern des Rechnerkerns, bestehend aus CPU, Bussystemen und Hauptspeicher. Viele Prüfverfahren, z.B. anwendungsspezifische Überprüfungen, setzen die Unversehrtheit dieses Systemkerns voraus. Einige der möglichen Fehlerquellen, z.B. Speicherfehler, können sehr effektiv durch fehlererkennende Codes abgedeckt werden. Die Paritätsprüfung im Hauptspeicher und verschiedene weitere Fehlererkennungstechniken sind mittlerweile in nahezu alle Rechnersysteme integriert. Mehrfachfehler und CPU-interne Fehler werden aber von der Paritätsprüfung nicht erkannt. Eine weitere Methode ist das Überwachen von Bustransaktionen mit Zeitschranken. Durch virtuelle Speicherverwaltung können Fehlzugriffe auf unerlaubte Adreßbereiche erkannt werden. Untersuchungen in [STDM82] zeigen, daß durch Überprüfung der Instruktionsadresse und der Datenadresse auf ihre Gültigkeit etwa 65% aller CPU-Fehler erkannt werden. Ein ähnliches Ergebnis ergibt sich in den Fehlerinjektionstests in Kapitel 4. Insgesamt ist damit zu rechnen, daß gängige Selbsttestmethoden etwa 60–80% aller temporären CPU-Fehler erkennen können. Eine gesamte Fehlerüberdeckung, die die 95%-Marke überschreitet, ist aber nur mit sehr hohem Aufwand erreichbar. Genaue Aussagen und Untersuchungen liegen bisher nicht vor.

Als Ergänzung der üblichen Techniken kann die Fehlererkennung durch weitere Hardwaremaßnahmen unterstützt werden. Beispiele sind der Einsatz von Watchdogprozessoren zur Kontrollflußüberwachung (siehe Kapitel 4) und andere Überwachungstechniken die einen Watchdogprozessor verwenden [MM88]. Beim Master-Checker-Betrieb [DHMP93] wird die Ausgabe zweier identischer Rechner verglichen um Fehler zu erkennen.

Auch anwendungsspezifische Techniken können verwendet werden. Dazu zählen Akzeptanztests, die zu bestimmten Zeitpunkten die errechneten Daten auf ihre Plausibilität überprüfen, beispielsweise auf die Einhaltung eines Konvergenzkriteriums.

Durch Invertierung kann ausgehend von einem Ergebnis eine mögliche Eingabe berechnet werden, die mit der tatsächlichen Eingabe verglichen wird. Weiterhin sind ECC-Verfahren für lineare Operationen bekannt. Zu den anwendungsspezifischen Tests in Betriebssystemen, die meistens von nahezu undurchschaubarer Komplexität sind, zählen semantische Überprüfungen interner Strukturen. Solche Überprüfungen sind aber für jede Applikation individuell zu erstellen.

Bei Betrachtung auf der Ebene der Rechnerknoten können die meisten dieser Maßnahmen zu den „Selbsttests“ gezählt werden: Die überprüfte Komponente überwacht sich selbst oder enthält Bestandteile, die sie überwachen, z.B. einen Watchdogprozessor. Beispiele für „Fremdtests“ sind die Zeitschrankenüberwachung von Lebenszeichen durch einen anderen Rechnerknoten, oder das Errechnen von Testsignaturen, die von einem anderen Rechner überprüft werden.

In verteilten Systemen ist eine kurze Latenz von Fehlern besonders wichtig, da sich Fehler, die zu spät erkannt werden, auf andere Komponenten ausbreiten können, beispielsweise durch fehlerhafte Ausgaben oder Nachrichten. Allgemein kann man daher sagen, daß Selbsttests eher geeignet sind um vorübergehende Fehler zu erkennen, während Fremdtests sich eher zum Erkennen dauerhafter Fehler eignen. Der Grund dafür liegt darin, daß mit Selbsttests eine konkurrente, enge Überwachung mit vergleichsweise geringeren Kosten möglich ist, da keine Daten zu einer anderen Einheit übertragen werden müssen. Dauerhafte Fehler, wie Systemabsturz oder Stromausfall, können aber nur durch Fremdtests erkannt werden. Da aber im Vergleich zu den vorübergehenden Fehlern dauerhafte Fehler um etwa eine Größenordnung seltener sind [Sie90], haben Fremdtests im Vergleich eine geringere Bedeutung.

2.4 Rückwärtsfehlerbehebung

Ein wichtiges Unterscheidungsmerkmal für Rückwärtsfehlerbehebungsverfahren ist die Transparenz. *Transparente* Verfahren erfordern keine Berücksichtigung bei der Programmerstellung, die Zustandssicherung und das Rücksetzen ist vielmehr ein nicht sichtbarer Systemdienst, der automatisch in regelmäßigen Abständen Sicherungspunkte erzeugt und im Fehlerfall die Applikation zurücksetzt.

Im Gegensatz dazu bestimmt der Programmierer bei der *programmgesteuerten* (nicht transparenten) Zustandssicherung selbst den Zeitpunkt und Umfang der Sicherungspunkterstellung. Es obliegt seiner Verantwortung dafür zu sorgen, daß Sicherungspunkte konsistent und in ausreichender Häufigkeit erstellt werden, und daß mit diesen Sicherungspunkten das Programm nach einem Fehler wieder starten kann. Die Sicherungspunkte bei programmgesteuerter Zustandssicherung benötigen oft erheblich weniger Speicherplatz und damit auch weniger Zeitaufwand für das Sichern. Die Vorgehensweise bei programmgesteuerter Zustandssicherung ist Gegenstand von Abschnitt 5.1.

2.4.1 Bestandteile eines Sicherungspunktes

Das Verhalten bzw. das Rechenergebnis einer Applikation wird durch Datenobjekte beeinflußt, die beim Zurücksetzen aus den Daten eines Sicherungspunkts restauriert werden müssen, um im anschließenden Wiederholungsbetrieb ein richtiges Ergebnis errechnen zu können. Zum Wiederherstellen des Zustands einer Applikation müssen dazu je nach Vorgehensweise (transparent oder programmgesteuert) unterschiedliche Informationen vorhanden sein, die die folgende Eigenschaft erfüllen:

> *Ein Sicherungspunkt enthält alle Daten, die eine Applikation benötigt, um (in Abhängigkeit von einem Rücksetzverfahren) nach dem Rücksetzen dasselbe Ergebnis wie im fehlerlosen Fall zu berechnen.*

Interaktion einer Applikation mit der Umgebung geschieht durch Manipulation von Objekten, auf die auch von außerhalb einer Applikation zugegriffen werden kann. Beim Rücksetzen einer Applikation können Konsistenzprobleme entstehen, wenn diese Objekte andere Vorgänge beeinflussen oder durch andere Vorgänge beeinflußt werden. In Datenbanksystemen werden deshalb Resourcen, die von außerhalb einer Kontrollsphäre zugänglich sind, durch Sperren geschützt, außerdem wird für veränderte Daten die Garantie übernommen, daß sie im Fehlerfall zurückgesetzt werden können. Analog können die Ausgaben einer Applikation erst dann als verbindlich anerkannt werden, nachdem die Applikation fehlerfrei terminiert hat. Allerdings entzieht sich das automatische Rücksetzen veränderter Objekte im Fehlerfall häufig der Kontrolle des Anwendungsprogramms: Beispielsweise ist es unmöglich, die Ausgabe eines Ergebnisses auf einem Zeilendrucker zurückzunehmen. Auch Eingaben, wie z.B. die Abfrage der aktuellen Uhrzeit, werden nach dem Rücksetzen andere Werte liefern, d.h. der Wiederholungslauf verhält sich nicht exakt wie der vorherige Programmlauf[1].

Die Anwendungen der Parallelrechner beschränken sich üblicherweise auf die Ein- und Ausgabe mittels Dateien. Dennoch ist auch hier meist keine Rücksetzmöglichkeit gegeben, da es wegen des dafür notwendigen Speicherbedarfs nicht praktikabel ist, Dateien oder ganze Dateisysteme in den Sicherungspunkt einer Applikation zu übernehmen. Immerhin gilt für Eingabedateien normalerweise, daß sie im Verlauf der Berechnung nicht verändert werden und nach dem Rücksetzen im selben Zustand vorgefunden werden. Der Fall der Ausgabedateien ist aber problematischer. Erfolgt die Ausgabe nur sequentiell in eine Ausgabedatei, so ist es möglich, den Schreibzeiger beim Rücksetzen wiederherzustellen. Alle nach dem Sicherungspunkt erzeugten, möglicherweise fehlerhaften Ausgabedaten werden dann beim Wiederholungslauf überschrieben. Wahlfreier Schreibzugriff bei der Ausgabe ist verboten,

[1] In [Pau88] wird für Datenbanksysteme versucht, Eingaben und Ausgaben in das Transaktionsmodell einzubeziehen.

oder er erfordert beim Rücksetzen eine Sonderbehandlung, die die Semantik der Applikation berücksichtigen muß[2].

Je nachdem, ob ein Objekt im Sicherungspunkt enthalten und damit wiederherstellbar ist oder nicht, werden im folgenden zwei Arten von Objekten unterschieden:

- interne Objekte: Registerinhalte, Daten-, Stack-, Codesegmente der beteiligten Prozesse, gemeinsame Speicherbereiche, Semaphore, Kommunikationskanäle
- externe Objekte (z.B. Dateien, Ein-/Ausgabegeräte, Uhren, ...)

Der Grad der Transparenz eines Rückwärtsfehlerbehebungsverfahrens ist dadurch bestimmt, an welcher Stelle die Grenze zwischen internen und externen Objekten gezogen wird. Allgemein gilt, daß völlige Transparenz eines Rückwärtsfehlerbehebungsverfahrens im allgemeinen Anwendungsfall nie erreichbar ist, vielmehr müssen dem Umgang einer Applikation mit der Außenwelt immer gewisse Beschränkungen auferlegt werden. Bei Mißachtung dieser Beschränkungen sind bei der Programmierung Sonderbehandlungen für den Fall des Rücksetzens erforderlich: Externe Objekte können sich nach dem Zurücksetzen aus Sicht der Applikation in einem inkonsistenten Zustand befinden.

Es gibt verschiedene Vorstellungen, wo die Grenze zwischen intern und extern zu ziehen ist. Insbesondere der Zustand des Rechners, speziell der des Betriebssystems, liegt in einer Grauzone. Je nach Betrachtungsweise ergeben sich verschiedene Bestandteile eines Sicherungspunktes. Bei transparenter Sicherung kann ein Prozeß zum Sichern an beliebigen Zeitpunkten unterbrochen werden, der gesamte Kontext ist also zwangsläufig wiederherzustellen, um einen Prozeß wieder am selben Punkt aufzusetzen. Besondere Probleme entstehen in Multitasking-Betriebssystemen wie z.B. UNIX: Teile des Prozeßkontext sind über viele interne Strukturen des Kerns verteilt, z.B. in der U-Area, in der Filedeskriptortabelle und in der Inode-Tabelle [Sie93]. Einträge in diesen Strukturen können gemeinsam mehreren nicht korrelierten Prozessen gehören, d.h. manche Objekte dürfen überhaupt nicht zurückgesetzt werden, da sonst die Arbeit anderer Anwendungen beeinflußt oder gestört wird.

Bei transparenter Zustandssicherung bietet sich oft eher ein Vorgehen auf Systemebene an. Im Fehlerfall werden alle momentan auf dem Rechnersystem laufenden Applikationen auf den letzten „Systemsicherungspunkt“ zurückgesetzt. Dieser Systemsicherungspunkt enthält den Zustand aller Applikationen sowie des Betriebssystems zum Zeitpunkt des Sicherns. Dieser Ansatz wird in [TS84] für Parallelrechner vorgeschlagen. Ein Nachteil ist, daß möglicherweise auch fehlerfreie Applikationen

[2] A. Reuter zitiert dazu in [Reu81] (S. 91) aus einem Manual des IBM System/360 Betriebssystems: „After taking a checkpoint, you must ensure that the data set contents are not changed in a manner that would make successful restart impossible“.

ebenfalls zurückgesetzt werden müssen. Die Rückwärtsfehlerbehebung als transparenter Dienst ist völlig entkoppelt von der Kenntnis über Struktur und Aufbau der Anwendungen. Pro Einzelrechner ist mit einem Speicherbedarf je Sicherungspunkt in der Größenordnung des verfügbaren Hauptspeichers zu rechnen.

Für die programmgesteuerte Zustandssicherung ist eine Betrachtung auf Anwendungsebene möglich. Der Zustand des Betriebssystems ist weitgehend ein externes Objekt. Im Fehlerfall ist die betroffene Anwendung zu identifizieren und beispielsweise durch ein Signal (vgl. Abschnitt 3.4.1) zu informieren. Nur die tatsächlich betroffene Applikation wird auf ihren jeweils letzten konsistenten Sicherungspunkt zurückgesetzt. Die Sicherungspunkte können sehr viel kleiner sein (15–50% des Speicherbedarfs eines Sicherungspunkts bei transparenter Zustandssicherung, vgl. Abschnitt 5.6). Nur die wesentlichen Daten werden gespeichert: Registerinhalte, Stackinhalt, sowie temporäre Daten sind ausgenommen, da das Programm an einem wohldefinierten Punkt fortgesetzt wird. Bei einem Systemabsturz kann das Betriebssystem nicht zurückgesetzt werden, sondern es muß durch einen Neustart in einen definierten Zustand versetzt werden (Vorwärtsfehlerbehebung). Erst dann wird die Applikation neu geladen und aufgesetzt.

Zum Verringern der Datenmenge ist eine inkrementelle Zustandssicherung möglich. Zum Sichern werden nur die Daten gespeichert, die seit dem letzten Sicherungspunkt tatsächlich verändert wurden. Dazu ist es notwendig, mit Hilfe der MMU die Speicherzugriffe eines Programms zu überwachen und eine Tabelle der veränderten Speicherseiten anzulegen. Da numerische Programme ohnehin innerhalb eines Berechnungsschritts schreibend auf alle sicherungsrelevanten Variablen im Arbeitsspeicher zugreifen, ist dieses Vorgehen im programmgesteuerten Ansatz nicht sinnvoll. Für die transparente Sicherung ist zu prüfen, inwieweit die zusätzlichen Kosten zur Verwaltung der Seitentabellen den möglichen Nutzen kompensieren.

2.4.2 Rückwärtsfehlerbehebung kooperierender Prozesse

Ein Anwendungsprogramm eines Parallelrechners wird üblicherweise unter der Modellvorstellung mehrerer kooperiender Prozesse betrachtet, die über Nachrichten miteinander kommunizieren[3]. In speichergekoppelten Systemen kann die Kommunikation über gemeinsame Variablen mittels Nachrichten modelliert werden, indem man den gemeinsamen Speicher einem Prozeß zuordnet und das Lesen oder Schreiben eines anderen Prozesses aus diesem Speicher als das Übertragen einer Nachricht betrachtet.

[3]Die duale Betrachtungsweise [SMR87], die Modellierung durch Objekte und Aktionen, ist für den Bereich der Datenbanksysteme verbreitet, im vorliegenden Themenkreis aber nicht so anschaulich.

Abbildung 2.2: Prozeßinteraktion

Wird nach dem Fehler des Prozesses P_1 dieser Prozeß auf den Sicherungspunkt SP_1 zurückgesetzt, so bedeutet das den Verlust der von Prozeß P_2 empfangenen Nachricht n. Wird umgekehrt, wenn Prozeß P_2 von einem Fehler betroffen ist, P_2 allein zurückgesetzt, so wird die Nachricht verdoppelt. Prozesse, die miteinander kommuniziert haben, sind also gleichzeitig zurückzusetzen, dabei sind jedoch die jeweiligen Rücksetzpunkte so zu wählen, daß ein *konsistenter* Systemzustand entsteht.

Die Definition der Konsistenz erfordert einen Ordnungsbegriff für Ereignisse wie „Nachricht n gesendet“ (z.B. $S_1(n)$) oder „Sicherungspunkt geschrieben“ (z.B. SP_1). In einem verteilten System kann man nur dann eine direkte Aussage über die Reihenfolge zweier Ereignisse treffen, wenn beide Ereignisse innerhalb desselben Prozesses auftreten. Alle anderen Ereignisse können nur indirekt durch die Interaktion von Prozessen geordnet werden. Entsprechend kann man die Ordnungsrelation „$\rightarrow$“ wie folgt definieren [Lam78]:

> *Seien a, b Ereignisse. Die Relation $a \rightarrow b$ (a geschieht vor b) ist wie folgt definiert:*
>
> *(1) Wenn Ereignisse a und b im selben Prozeß auftreten und a vor b stattfindet, so gilt $a \rightarrow b$*
>
> *(2) Wenn das Ereignis a das Senden einer Nachricht und b das Empfangen derselben Nachricht ist, so gilt $a \rightarrow b$.*
>
> *(3) Gilt $a \rightarrow b$ und $b \rightarrow c$, so gilt $a \rightarrow c$.*
>
> *(4) Zwei verschiedene Ereignisse a und b finden gleichzeitig statt, wenn gilt $a \not\rightarrow b$ und $b \not\rightarrow a$.*

Abbildung 2.3: Rücksetzlinien

2.4.2.2 Rücksetztrennung

Ein weiteres Problem der Nachrichtenkonsistenz kann auftreten, wenn sich alte Nachrichten (vor dem Rücksetzen erzeugt) mit neuen Nachrichten (nach dem Rücksetzen erzeugt) vermischen. Eine Fehlermöglichkeit ist beispielsweise, daß eine Nachricht, die vor dem Rücksetzen bereits abgeschickt wurde, von einem zurückgesetzten Prozeß empfangen wird. Umgekehrt darf keine neue Nachricht von einem noch nicht zurückgesetzten Prozeß empfangen werden. Solche Fehler können aus zwei Gründen

Abbildung 2.4: Inkonsistenz beim Wiederanlauf

In Abbildung 2.4a erleidet der Prozeß P_2 einen Fehler und setzt zurück auf seinen Sicherungspunkt SP_2. P_1 muß ebenfalls zurückgesetzt werden, da beide Prozesse kommuniziert haben. Es verstreicht jedoch etwas Zeit, bis P_1 den Fehler von P_2 bemerkt. In dieser Zeit (Abbildung 2.4b) fährt P_2 mit der Ausführung des Programms fort und sendet wiederum seine Nachricht an den Prozeß P_1. P_1 empfängt diese Nachricht ein zweites Mal, setzt daraufhin auf SP_1 zurück und verliert diese Nachricht. Als Abhilfe können alle Prozesse vor dem Start synchronisiert und alte Nachrichten vernichtet werden.

Das gleiche Problem besteht in speichergekoppelten Systemen. Enthält der Sicherungspunkt Daten, die den gemeinsamen Speicher wiederherstellen, so muß beachtet werden, daß die Synchronisation und Freigabe zum Schreiben durch andere Prozesse erst nach dem Laden dieser Daten erfolgt. Ansonsten kann ein anderer Prozeß ungültige Daten lesen oder Daten in den gemeinsamen Speicher schreiben, die nachher beim Laden der Sicherungsdaten wieder überschrieben werden.

2.4.2.3 Der Dominoeffekt

In Abschnitt 2.4.2.1 wurde ein Kriterium vorgestellt, anhand dessen die Sicherungspunkte eines Prozeßsystems auf ihre Konsistenz überprüft werden können. Für die Zustandssicherung können nun zwei verschiedene Strategien unterschieden werden:

Unabhängige Zustandssicherung. Hierbei kann jeder Prozeß für sich selbst entscheiden, wann er einen Sicherungspunkt speichert. Im Fehlerfall wird anhand des Konsistenzkriteriums aus Abschnitt 2.4.2.1 überprüft, welche dieser Sicherungspunkte zueinander passend sind, um einen (möglichst aktuellen) konsistenten Zustand zu restaurieren.

Konsistente Zustandssicherung. Bei dieser Strategie koordinieren die Prozesse die Zeitpunkte, zu denen sie Sicherungspunkte erstellen, so daß die Menge der aktuellen Sicherungspunkte immer konsistent bleibt.

Abbildung 2.5: Dominoeffekt

Bei der konsistenten Zustandssicherung kann im Fehlerfall schneller zurückgesetzt werden, da eine konsistente Sicherungspunktmenge nicht erst gesucht werden muß. Darüberhinaus muß bei konsistenter Zustandssicherung nicht eine ganze Vorgeschichte von Sicherungspunkten gespeichert werden, sondern nur der jeweils aktuellste Sicherungspunkt. Allerdings dürfen „alte“ Sicherungspunkte erst dann gelöscht werden, wenn auch tatsächlich ein neuerer Sicherungspunkt vorhanden ist, auf den konsistent zurückgesetzt werden kann.

2.4.3 Zustandssicherungsverfahren

Die üblichen Strategien der konsistenten Zustandssicherung können nach drei grundlegenden Techniken eingeteilt werden: Nachrichtenaufzeichnung, Konversationen und Rücksetzlinienausbreitung. Die globale Zustandssicherung kann als Spezialfall der Rücksetzlinienausbreitung betrachtet werden.

Nachrichtenaufzeichnung. Dieses Verfahren (z.B. [BBG83, PP83]) erlaubt nahezu unabhängige Zustandssicherung und Wiederanlauf. Sämtliche Nachrichten, die ein Prozeß sendet oder empfängt, werden aufgezeichnet. Nach dem Rücksetzen können die seit dem letzten Sicherungspunkt empfangenen Nachrichten wieder abgespielt werden. Die Schwierigkeiten bei der Nachrichtenaufzeichnung sind Determinismus und Validierung: Das Programm muß sich deterministisch verhalten, also beim Wiederholungslauf dieselben Nachrichten versenden wie im vorherigen

Programmlauf, soweit der vorherige Programmlauf fehlerfrei war. Beim Validierungsproblem stellt sich die Frage, ob eine empfangene Nachricht vertrauenswürdig ist. Weichen die Nachrichten, die im Wiederholungsbetrieb von einem Prozeß gesendet werden, von den vorher gesendeten Nachrichten ab (ein Fehler tritt vor dem Versenden einer Nachricht auf, wird aber erst danach erkannt), so liegt ein Folgefehler vor. Die Nachricht wird im nachhinein für ungültig erklärt und der Empfänger zurückgesetzt. Diese Kette kann sich nur soweit, wie sich ein Fehler ausgebreitet hat, fortsetzen, d.h. der Dominoeffekt tritt nicht auf.

Man kann bei der Nachrichtenaufzeichnung zwischen pessimistischen und optimistischen Verfahren unterscheiden. Der (oben beschriebene) pessimistische Ansatz bevorzugt schnelle Fehlererholung. Da Fehler meistens nur sehr selten auftreten, versuchen optimistische Verfahren (z.B. [SY85]) den Aufwand der Nachrichtenaufzeichnung im fehlerfreien Fall auf Kosten des Aufwands im Fehlerfall zu reduzieren.

Der Vorteil dieses Verfahrens ist es, daß nur die tatsächlich von einem Fehler und seinen Folgen betroffenen Prozesse zurückgesetzt werden müssen. Allerdings ist das Abspeichern der Nachrichten bei jeder Kommunikation sehr aufwendig. Insbesondere bei speichergekoppelten Rechnersystemen wird der Vorteil gemeinsamen Speichers durch den Aufzeichnungszwang vernichtet.

Konversationen [Ran75, RT79]. In diesem Ansatz werden durch „Konversationen“ die Kommunikationsbeziehungen für ein Zeitintervall vom Programmierer deklariert. Beim Betreten einer Konversation speichert ein jeweils beteiligter Prozeß einen Sicherungspunkt. Falls einer der an einer Konversation teilnehmenden Prozesse ausfällt, so setzen alle Teilnehmer auf den Sicherungspunkt zurück. Innerhalb einer Konversation dürfen die beteiligten Prozesse beliebig kommunizieren, nicht aber mit anderen Prozessen („Information Smuggling“). Es handelt sich um ein nicht-transparentes Verfahren[4].

Rücksetzlinienausbreitung [BS83, CL85, KYA86, KT87]. Bei dieser Methode wird sichergestellt, daß alle Prozesse, die miteinander kommuniziert haben, eine gemeinsame Rücksetzlinie besitzen. Dazu darf zwischen dem Speichern der lokalen Sicherungspunkte keine Kommunikation zwischen den beteiligten Prozessen stattfinden. Ein Bestandteil des Sicherungspunktes sind möglicherweise die Nachrichtenpuffer oder, im Falle von Speicherkommunikation, gemeinsame Speicherbereiche. Um zu erkennen, welche Prozesse miteinander kommuniziert haben, muß das Kommunikationssystem Aufzeichnungen über die Kommunikationsbeziehungen speichern. Wird beispielsweise in [KT87] die Zustandssicherung angestoßen, kann anhand dieser Information entschieden werden, welche Prozesse sich am Sichern beteiligen müssen. Dabei ist die Relation „P_i hat mit P_j kommuniziert“ transitiv, d.h.

[4]Für das Konzept der Konversationen kann eine enge Verwandtschaft mit den Fehlerbehandlungsverfahren aus dem Bereich der Datenbanktechnik gesehen werden [SMR87]. Diese Verfahren erfordern eine Strukturierung der Applikation nach Gesichtspunkten der Transaktionsverarbeitung. Eine solche Strukturierung ist für den betrachteten Anwendungsfall nicht geeignet.

wenn P_j mit einem weiteren Prozessor P_k kommuniziert hat, so muß, wenn P_i einen Sicherungspunkt speichert, außer P_j auch P_k einen Sicherungspunkt speichern.

Globale Zustandssicherung. Wenn man für den Fall der Rücksetzlinienausbreitung davon ausgeht, daß bis zum Schreiben eines Sicherungspunktes ohnehin immer alle Prozessoren miteinander kommuniziert haben, so kann man sich Aufzeichungen über die Kommunikationsbeziehungen sparen und kommt zur globalen Zustandssicherung [TS84]: Das Verwalten der Kommunikationsbeziehungen erfordert nur zusätzlichen Aufwand. Für die typischen Applikationen eines Hochleistungsparallelrechners ist dieser Fall gegeben (vgl. Abschnitt 2.1.3). In ähnlicher Weise würde im Konzept der Konversationen nur eine globale Konversation sinnvoll sein. Auch im DIRMU-System [HMW85] wurde globale Zustandssicherung eingesetzt, sowie im transputerbasierten Nachfolgermodell DAMP [BM91].

Bei einer transparenten Implementierung der globalen Zustandssicherung müssen zum Sichern alle Prozesse einer Anwendung angehalten werden. Erst nach dem Eintreffen aller Nachrichten, die eventuell noch unterwegs sind, wird ein globaler Rücksetzpunkt erstellt. Durch diese Synchronisationspause entsteht ein geringer Leistungsverlust. Bei programmgesteuerter, globaler Zustandssicherung muß der Programmierer geeignete Zeitpunkte identifizieren, an denen keine Kommunikation der Prozesse stattfindet, möglicherweise müssen alle beteiligten Prozesse zur Zustandssicherung explizit synchronisiert werden.

Basierend auf Kombinationen dieser Grundkonzepte wurden verschiedene Hybridverfahren vorgestellt. Beispielsweise werden in [LNP91, CJ91] Kombinationen aus globaler Zustandssicherung und Nachrichtenaufzeichnung untersucht. Zur globalen Zustandssicherung werden nicht alle Prozesse angehalten und synchronisiert, sondern es wird damit begonnen, die Nachrichten solange aufzuzeichnen, bis alle Prozesse einen Sicherungspunkt erstellt haben. Dadurch wird die fehlende Nebenläufigkeit der globalen Zustandssicherung vermieden; auch der hohe Zeitaufwand der Nachrichtenaufzeichung tritt nur in einem kurzen Intervall während der Zustandssicherung auf.

Wegen der in Abschnitt 2.4.2.3 genannten Gründe und der typischen Eigenschaften von Applikationen (Abschnitt 2.1.3) kommen für Hochleistungsparallelrechner nur Verfahren zur konsistenten Zustandssicherung in Frage, und von diesen Verfahren wiederum nur die globale Zustandssicherung. Ein Nachteil des Ansatzes ist allerdings die verringerte Nebenläufigkeit wegen der notwendigen Synchronisierung beim Schreiben der Sicherungspunkte. Wie auch in Abschnitt 5.1 noch zu sehen ist, enthalten numerische Applikationen natürliche Synchronisationspunkte, an denen es mit geringem Verlust an Nebenläufigkeit möglich ist, Sicherungspunkte zu speichern. Auch die in Kapitel 5 beschriebenen Verfahren werden versuchen, diesen Nachteil weitgehend zu mildern.

Nachrichtenaufzeichnung ist wegen des hohen Kommunikationsaufkommens in Parallelrechnern ungeeignet, denn entgegen einer weitverbreiteten Vorstellung besteht

für parallele Applikationen eine obere Grenze für den maximal erreichbaren Speedup. Über Amdahls Gesetz hinaus entstehen zusätzliche Verluste durch Kommunikation, die bei genügend hoher Prozessorzahl den erreichten Speedup sogar wieder sinken lassen. Durch aufwendige Kommunikationsalgorithmen, wie sie beispielsweise für Verfahren der Nachrichtenaufzeichnung notwendig sind, würde dieses besonders im Bereich der massiv parallelen Rechner unerwünschte Verhalten noch verstärkt. Abgesehen davon ist für Systeme mit Speicherkommunikation Nachrichtenaufzeichnung mit sehr hohem Mehraufwand verbunden und deshalb überhaupt nicht praktikabel. Ein Hybridverfahren aus globaler Zustandssicherung und Nachrichtenaufzeichnung, wie in [LNP91, CJ91], ist aber für reine Message-Passing Systeme durchaus geeignet. Wegen der Speicherkopplung kann man dieses Verfahren nicht für MEMSY verwenden.

Kapitel 3

Anforderungen in Multiprozessoren

Die Fehlertoleranzmaßnahmen zur Rückwärtsfehlerbehebung können in drei aufeinander aufbauende Bereiche unterteilt werden: Fehlererkennung, Zustandssicherung, sowie Fehlerbehebung (vgl. Abschnitt 2.2). Der Einsatz von Fehlertoleranz in diesen Bereichen ist mit zusätzlichem Aufwand verbunden, aber bei zunehmender Komplexität und zunehmendem Zeit- und Hardwareaufwand der Maßnahmen wird keineswegs eine proportional ansteigende Verbesserung der Fehlertoleranzeigenschaften eines Multiprozessors erreicht. Komplexe, aufwendige Maßnahmen, wie z.B. die automatische Rekonfiguration des Rechners nach einem dauerhaften Ausfall, sind einerseits mit hohem Aufwand (zusätzliche Verbindungswege, Ersatzkomponenten) verbunden, werden andererseits aber nur in vergleichsweise seltenen Fällen beansprucht. Folglich ist vor dem Entwurf von Rückwärtsfehlerbehebungsmaßnahmen zu klären, welche Funktionalität im Bereich der Hochleistungsparallelrechner sinnvoll ist und nach welchen Kriterien diese Entscheidung getroffen werden kann.

3.1 Bewertungsgrößen

Die folgenden zwei Eigenschaften können als Bewertungsmaßstab für die Fehlertoleranzeigenschaften eines Multiprozessors verwendet werden (vgl. [DGH+93]):

- Die **Effizienz** $E(t)$ der Fehlertoleranzmaßnahmen ist ein Maß für die bei der Bearbeitung eines (repräsentativen) Programms der Rechendauer t aufgebrachte (Zeit)-Redundanz.

- Die **Vertrauenswürdigkeit** $V(t)$ gibt an, mit welcher Wahrscheinlichkeit ein Rechenergebnis nach Ablauf der Rechenzeit t fehlerfrei ist.

Die Effizienz errechnet sich aus dem Verhältnis der Rechenzeit T_0, die in einem idealen fehlerlosen System ohne Fehlertoleranzmaßnahmen für die Berechnung notwendig wäre, und der in einem realen System mit Fehlertoleranz, in dem Fehler auftreten können, im Mittel verbrauchten Rechenzeit T_{FF}:

$$E(T_0) = \frac{T_0}{T_{FF}} \tag{3.1}$$

Neben dem Zeitaufwand für die Fehlerbehebung, während der kein Nutzbetrieb geleistet wird, berücksichtigt die Definition der Effizienz auch den Zeitverlust für das Speichern der Sicherungspunkte. Weitere Einflußfaktoren werden im Verlauf dieses Kapitels vorgestellt[1].

Abbildung 3.1 zeigt ein einfaches Zustandsdiagramm eines Systems mit Rückwärtsfehlerbehebung. Im fehlerfreien Betrieb befindet sich das System in einem der beiden Zustände „Arbeiten“ oder „Sichern“. Das Sicherungsintervall sei eine konstante Zeitdauer von t_i. Der Zeitbedarf für das Speichern eines Sicherungspunkts t_s sei ebenfalls konstant. Die Dauer eines Arbeitsabschnitts t_i zuzüglich der Sicherungszeit t_s wird im folgenden als „Sicherungszyklus“ bezeichnet. Das Auftreten eines Fehlers sei exponentiell verteilt mit der Fehlerrate λ ($1/\lambda$ = MTTF). Wird ein Fehler erkannt, so geht das System in den Zustand „Rücksetzen“. Durch das Rücksetzen geht der bis dahin im Sicherungsintervall angefallene Rechenfortschritt t_v verloren. Zusätzlich kann eine konstante Erholungs- und Reaktionszeit t_r angenommen werden.

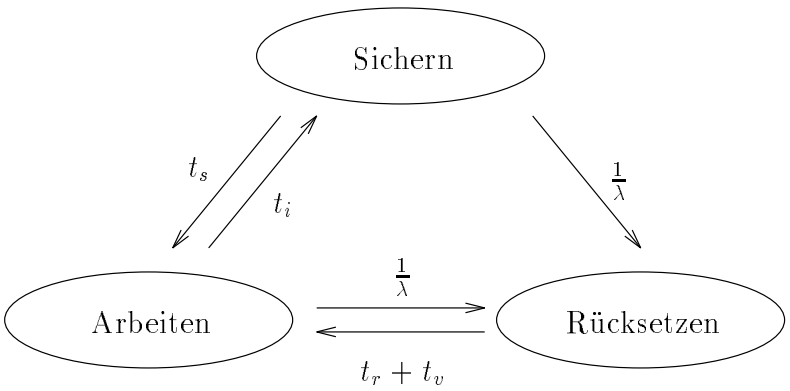

Abbildung 3.1: Zustände eines Systems mit Rückwärtsfehlerbehebung

Die folgenden Überlegungen basieren auf diesem einfachen Modell, obwohl nicht alle tatsächlich möglichen Zustände dargestellt sind. In realen Systemen wird der Fall

[1]Die Effizienz könnte auch als die Wahrscheinlichkeit dafür definiert werden, daß ein System zu einem Zeitpunkt t für den Nutzbetrieb *verfügbar* ist. Um Verwechslungen mit der Verfügbarkeit im Sinne der üblichen Definition zu vermeiden (als die Wahrscheinlichkeit dafür, daß das System zu einem Zeitpunkt t die Anforderungen erfüllen kann [Gör89]), wird der Begriff „Effizienz“ verwendet.

auftreten, daß ein Fehler überhaupt nicht oder erst nach einer Verzögerung erkannt wird. Auch die Vorgänge in einem Multiprozessor sind nicht berücksichtigt. Vielmehr hat man im Multiprozessor eine Menge von mehr oder weniger autonomen Teilsystemen, die durch Koordinationsprotokolle (Kapitel 6) gesteuert die Phasen und Zustände des obigen Modells mehr oder weniger gleichzeitig durchlaufen müssen. Grobe Unterschiede im Ablauf, beispielsweise wenn eines der Teilsysteme zurückgesetzt wird und die anderen nicht, führen zu Konsistenzproblemen. Trotzdem ist dieses Modell der Ausgangspunkt für die folgenden Betrachtungen: Zur Bestimmung von Formeln für Effizienz und Vertrauenswürdigkeit, sowie zur Bestimmung des Gewichts verschiedener Einflußgrößen.

Für die folgenden Zahlenbeispiele wurde als MTTF eines Einzelrechners ein Jahr angenommen. Für einen Parallelrechner mit 1000 Prozessoren ergibt sich damit eine MTTF von $1/\lambda \approx 8$ Stunden. Für Anwendungsprogramme, die ein stabiles Konvergenzverhalten haben, könnte eine geringere Rate als im allgemeinen Fall angesetzt werden. Diese Verfahren verhalten sich häufig gegenüber eingestreuten Fehler indifferent, d.h. ein hoher Anteil der Fehler im Datenbereich wird in weiteren Iterationsschritten wieder ausgemittelt. Welche Fehlerrate tatsächlich anzusetzen ist, kann aber nur für Einzelfälle nach langjähriger Beobachtung bestimmt werden. Die Zahlenbeispiele beziehen sich deshalb auf weniger robuste, „normale“ Applikationen.

3.2 Fehlererkennung

Der Begriff Fehlerdiagnose in einem Multiprozessor umfaßt zwei Funktionen: Zum einen das Erkennen von Fehlern, zum anderen das Herstellen einer einheitlichen Systemsicht mit dem Ziel, defekte Komponenten zu lokalisieren und auszugrenzen [DGT86]. Letztere Funktionen sind aber nur im Falle von automatischem Wiederanlauf und dynamischer Rekonfiguration wichtig. Die Fehlerlokalisierung soll deshalb als ein Bestandteil der Fehlerbehebung betrachtet werden (Abschnitt 3.4).

3.2.1 Einfluß der Fehlerüberdeckung

In Abschnitt 2.3 wurde bereits erwähnt, daß in realen Systemen nie alle Fehler erkannt werden. Vielmehr können „fatale“ Fehler auftreten: Wird ein Fehler nicht erkannt, so endet die Berechnung mit einem falschen Ergebnis. Die Fehlerüberdeckung beeinflußt folglich die Vertrauenswürdigkeit des Rechenergebnisses.

Maß für die Fehlerüberdeckung ist der Coverage-Faktor c, der den Anteil der Fehler angibt, die erkannt werden. Unerkannte Fehler müssen nicht immer zu einem fehlerhaften Programmlauf führen, denn wenn im selben Sicherungszyklus ein anderer Fehler auftritt und erkannt wird, werden sie durch Rücksetzen ebenfalls behoben.

Dieser Einfluß ist aber gering, weil meistens die MTTF $(1/\lambda)$ als groß gegenüber der Länge eines Sicherungszyklus angenommen werden kann $(1/\lambda \gg t_i + t_s)$. Die Rate, mit der nicht tolerierbare Fehler auftreten, errechnet sich dann zu $(1-c)\cdot\lambda$. Für die Vertrauenswürdigkeit $V(t)$ ergibt sich damit:

$$V(t) = e^{-(1-c)\lambda t} \tag{3.2}$$

Abbildung 3.2 zeigt die Wahrscheinlichkeit dafür, daß ein Programm nicht von einem unerkannten Fehler betroffen ist, in Abhängigkeit von der Programmlaufzeit. Soll für das bereits vorher betrachtete Beispielsystem $(1/\lambda \approx 8$ Stunden) bei einer Fehlerüberdeckung von 95% ein mit 90%-iger Wahrscheinlichkeit korrektes Ergebnis errechnet werden, so darf ein Programm nur etwa eine Laufzeit von $T_{FF} \approx 16$ Stunden haben.

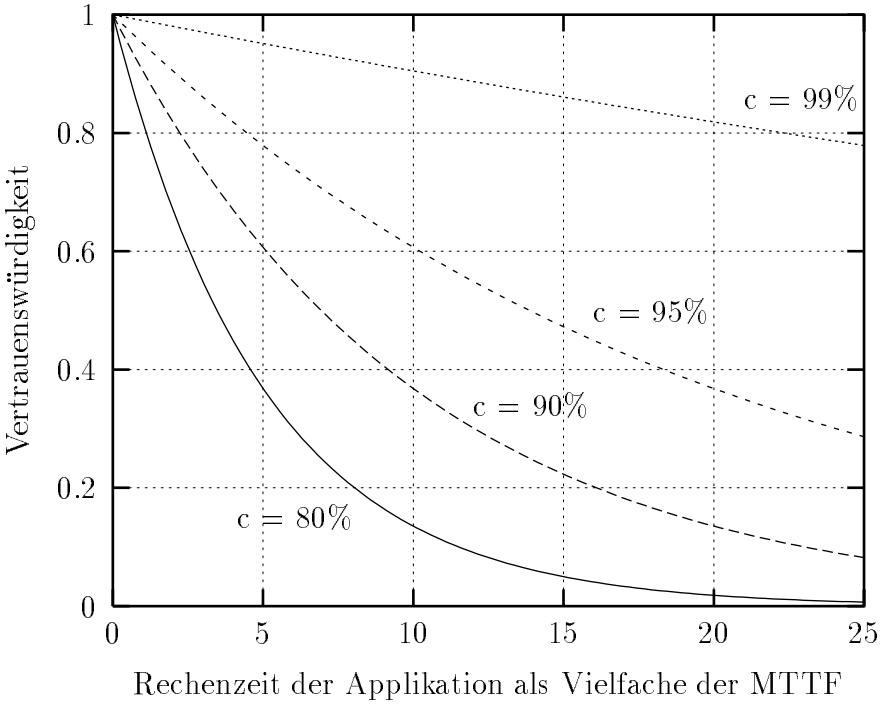

Abbildung 3.2: Einfluß nicht perfekter Fehlererkennung

Die 1000 Prozessoren des Beispielsystems leisten innerhalb der 16 Stunden eine Rechenarbeit von 16000 Prozessorstunden, um ein Ergebnis zu errechnen, das mit einer Wahrscheinlichkeit von 90% korrekt ist. Die Obergrenze an Prozessorstunden, bei der ein Ergebnis mit einer gewissen Wahrscheinlichkeit korrekt ist, hängt dabei nicht von der Prozessorzahl ab, da, unter der Annahme unkorrelierter Fehler,

einer n-fachen Leistung (idealer Speedup[2]) eine n-fache Fehlerrate gegenübersteht[3]. Durch massiv parallele Rechner kann zwar die Wartezeit, um ein Ergebnis zu errechnen, reduziert werden, die Obergrenze für die Durchführbarkeit einer aufwendigen Berechnung ist aber weiterhin durch die Fehlerrate eines Einzelrechners und durch die Fehlerüberdeckung bestimmt.

3.2.2 Einfluß der Fehlerlatenz

Auch durch die Latenzzeit, die bis zum Erkennen eines Fehlers verstreicht, können Schwierigkeiten auftreten. Wird ein Fehler e, der vor dem Schreiben des Sicherungspunktes eines beteiligten Prozesses P_i aufgetreten ist, erst nach dem Schreiben des Sicherungspunktes erkannt, so ist, sofern immer nur der letzte Sicherungspunkt aufbewahrt wird, das Rücksetzen nicht möglich. Wenn eine physikalische Fehlerursache in der Lage ist, ein falsches Rechenergebnis zu bewirken, so muß sich diese Fehlerursache auch auf die Daten im Sicherungspunkt auswirken, da ein Sicherungspunkt nach Definition alle Daten umfaßt, die das Ergebnis beeinflussen.

Nach dem Rücksetzen auf den letzten Sicherungspunkt wird also derselbe Fehler wieder auftreten: Der Rechenauftrag muß abgebrochen und von Anfang an wiederholt werden. Die Latenz der Fehlererkennung wirkt sich folglich auf die Effizienz der Rückwärtsfehlerbehebung aus. Der Einfluß verstärkt sich mit zunehmender Dauer eines Rechenauftrags, da bei langen Berechnungen auch lange Rechenzeiten durch Neustart verloren gehen können. Die Zeit, die eventuell durch vergebliche Rücksetzversuche auf den fehlerhaften Sicherungspunkt entsteht, soll demgegenüber vernachlässigt werden. Auch der Fall, daß beim Terminieren des Rechenauftrags noch ein unerkannter Fehler vorliegt, soll nicht betrachtet werden ($1/\mu \ll T_0$). Werden weiter zurückliegende Sicherungspunkte aufbewahrt, so verringert sich der Einfluß der Latenz auf die Effizienz.

Unter der Annahme, daß zum Zeitpunkt des Sicherns latente Fehler mit einer Rate λ_l auftreten, beträgt der Erwartungswert für die Zeit, nach der die Rechnung erfolgreich abgeschlossen ist[4]:

$$E[T] = \frac{1}{\lambda_l}(e^{+\lambda_l T_0} - 1) \tag{3.3}$$

Durch Einfluß der Fehlerlatenz verringert sich damit die Effizienz der Rückwärtsfehlerbehebung nach der Formel

[2] In realen Systemen sinkt die Effizienz der numerischen Verfahren mit zunehmendem Grad der Parallelität.

[3] Falls der Speicherbedarf der Applikation trotz zunehmender Knotenzahl konstant bleibt, so muß für Speicherfehler nicht mit einer n-fachen Fehlerrate gerechnet werden. Im Hinblick auf Speicherfehler kann ein größerer Multiprozessor wegen der kürzeren Rechenzeit sogar zuverlässiger sein [Hes89].

[4] Zur Herleitung siehe Anhang A.1.

$$E_L(T_0) = \frac{T_0}{T} = \frac{\lambda_l T_0}{e^{\lambda_l T_0} - 1} \tag{3.4}$$

Die Rate, mit der latente Fehler auftreten, hängt von der Länge des Sicherungszyklus und von der Verteilung der Fehlerlatenzzeiten ab. Bei der Modellierung durch Markoff-Prozesse wird von einer exponentiellen Verteilung der Latenzzeiten mit einer Rate μ ausgegangen. Unter dieser Annahme errechnet sich die Wahrscheinlichkeit $P_{lat}(t)$, daß ein Fehler innerhalb eines Zeitraums der Länge t nicht erkannt wird, nach der Formel:

$$P_{lat}(t) = e^{-\mu t} \tag{3.5}$$

Unter der Annahme, daß das Auftreten eines Fehlers zu jedem Zeitpunkt eines Sicherungszyklus gleich wahrscheinlich ist ($1/\lambda \gg t_i + t_s$), errechnet sich die Wahrscheinlichkeit l dafür, daß am Ende eines Sicherungszyklus ein latenter Fehler existiert zu:

$$l = \frac{1}{t_i + t_s} \int_0^{t_i+t_s} e^{-\mu t} dt = \frac{1 - e^{-\mu(t_i+t_s)}}{\mu(t_i + t_s)} \tag{3.6}$$

Im praktischen Betrieb wird man fordern, daß diese Wahrscheinlichkeit sehr klein ist. In diesem Bereich ($1/\mu \ll t_i + t_s$) kann $e^{-\mu(t_i+t_s)} \approx 0$ angenommen werden, d.h. es gilt näherungsweise

$$l \approx \frac{1}{\mu(t_i + t_s)} \tag{3.7}$$

Für eine realistischere Modellierung können, je nach Fehlererkennungsmethode, die Latenzzeiten auch als konstant oder auf irgendeine andere Weise verteilt angenommen werden. Das obige Ergebnis entspricht genau dem Verhältnis aus Fehlerlatenzzeit und Sicherungszyklus, das sich für eine konstante Latenzzeit der Dauer $1/\mu$ ergeben würde.

Unter der Annahme, daß tatsächlich alle Fehler irgendwann erkannt werden ($c \approx 1$), kann in Formel 3.4 die Rate

$$\lambda_l \approx l\lambda \approx \frac{\lambda}{\mu(t_i + t_s)} \tag{3.8}$$

angesetzt werden. Werden die Sicherungspunkte über mehrere Sicherungszyklen aufbewahrt, so ist $t_i + t_s$ jeweils mit einem entsprechenden Faktor zu multiplizieren.

Im bisherigen Zahlenbeispiel ($1/\lambda \approx 8$ Stunden) sei nun ein Sicherungszyklus von etwa 31 Minuten ($t_s + t_i \approx 0{,}065/\lambda$) angenommen. Dieser Wert ergibt sich bei

Wahl des optimalen Sicherungsintervalls (im Vorgriff auf Abschnitt 3.3.2) bei einer angenommenen Speicherzeit von etwa einer Minute ($t_s = 0{,}002/\lambda$). Damit der Mehraufwand bei einer Rechendauer von 16 Stunden ($T_0 = 2/\lambda$) die 5%-Marke nicht übersteigt, muß die mittlere Fehlerlatenz kleiner als etwa 100 Sekunden ($\mu \approx 300\lambda$) sein.

In Abbildung 3.3 ist die Effizienz für verschiedene Werte der Wahrscheinlichkeit l aufgetragen. Die Laufzeit des Programms wurde wiederum auf $1/\lambda$ normiert. Zum Vergleich wurde auch die Kurve für einen Rechner ohne Rückwärtsfehlerbehebung ($l = 1$, bei jedem Fehler muß vom Anfang gestartet werden) aufgetragen. Für das Zahlenbeispiel gilt $l = \frac{1}{\mu(t_i+t_s)} = 0{,}05$.

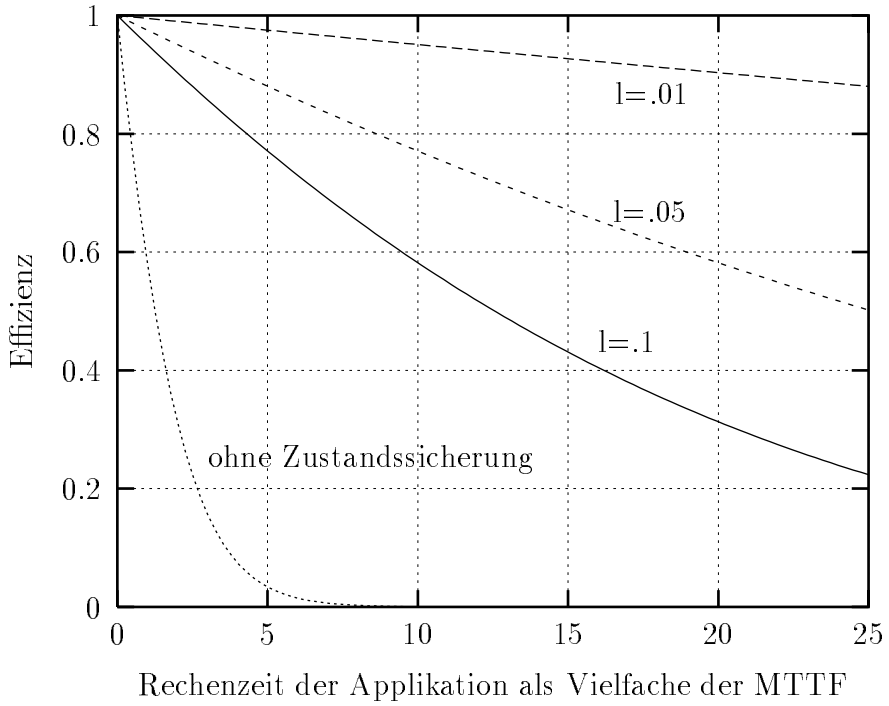

Abbildung 3.3: Effizienz je nach Wahrscheinlichkeit eines Neustarts

Weitere, ähnliche Einflüsse entstehen durch nichtideales Verhalten des Sicherungsspeichers (siehe Abschnitt 3.3.1) und durch mangelnde Koordination bei der Zustandssicherung (Abschnitt 3.3.3).

3.3 Zustandssicherung

Die Dauer der Zustandssicherung t_s beeinflußt maßgeblich die Effizienz der Rückwärtsfehlerbehebung. Schnelles Speichern der Sicherungspunkte ermöglicht es, ein

kürzeres Sicherungsintervall t_i zu wählen, und damit den Rechenzeitverlust beim Rücksetzen zu verringern. Die Zeitdauer t_s ist bestimmt durch den Umfang eines Sicherungspunkts, die Eigenschaften des Speichermediums und durch die Vorgehensweise beim Sichern.

3.3.1 Speichermedien und Sicherungszeit

Die Idealvorstellung eines nichtflüchtigen Speichers wurde in [Lam88] unter dem Begriff stabiler „Speicher" eingeführt. Gleichzeitig wurde eine Implementierung stabilen Speichers auf Festplattenlaufwerken vorgestellt, die durch das Verwenden zweier Kopien der Daten (evtl. auf physikalisch getrennten Platten) und Operationen wie „careful read" und „careful write" versucht, dem Ideal möglichst nahe zu kommen. Auch Umsetzungen dieses Konzepts mithilfe von RAM-Bausteinen sind geplant oder wurden bereits vorgenommen [BMRS91, DGH+93]. In [Leh90] wurden ebenfalls RAM-Speicher für die Sicherungspunkte verwendet. Die Sicherungspunkte wurden in diesem Verfahren in mehreren benachbarten Knotenrechnern redundant gespeichert. Gegen den Einsatz RAM-basierter stabiler Speicher in großen Parallelrechnern sprechen aber vor allem die gegenwärtig im Vergleich zur Festplatte hohen Kosten von RAM-Bausteinen (2 bzw. 40 Dollar/MB), zumal die schnelle, wahlfreie Zugriffsmöglichkeit von RAM-Bausteinen bei einer Verwendung als Sicherungsspeicher nicht ausgenutzt wird.

Ein weiterer interessanter Ansatz sind redundante Festplattensysteme (RAID, redundant array of inexpensive disks) [GKP88]. Einfache, billige Festplatten sind quasi parallel geschaltet. Eine fehlertolerierende Codierung erlaubt es, Ein- oder Mehrbitfehler zu korrigieren und defekte Einzelplatten im laufenden Betrieb auszutauschen. Mehrere verschiedene RAID-Versionen mit unterschiedlichen Eigenschaften (Zugriffslatenz, Transferrate, Datensicherheit) sind im Handel.

Durch ein Fehlverhalten des Sicherungsspeichers können ebenfalls Fehler ausgelöst werden, z.B. wenn der aktuelle Sicherungspunkt eines der beteiligten Prozesse durch einen Fehler des Sicherungsspeichers zerstört ist und zurückgesetzt werden muß. Die Fehlerrate eines Sicherungsspeichers hat also ebenfalls eine Auswirkung auf die Effizienz der Rückwärtsfehlerbehebung.

Für ein Festplattenlaufwerk wird in einer aktuellen Marktübersicht je nach Hersteller eine MTTF von Hunderttausend bis zu einer Million Betriebsstunden (11 bis 114 Jahre) angegeben. Wird ein Festplatte als Sicherungsspeicher verwendet, so werden dauerhafte Ausfälle die Effizienz nicht wesentlich beeinflussen. Etwa alle 10^{12} bis 10^{13} gelesene Bits ist aber mit einem nicht korrigierbaren Lesefehler zu rechnen [Bäh91]. Bei Sicherungspunkten der Größe 10 MB (etwa 10^8 Bit) pro Prozessor ist ein Zugriff auf einen Sicherungspunkt unter 10000 durch einen Fehler betroffen. Je nach Prozessorzahl n wird also jedes $\frac{10000}{n}$-te Rücksetzen nicht gelingen. Der Faktor l für Formel 3.8 errechnet sich also zu:

$$l = \frac{n}{10000} \tag{3.9}$$

Für das Beispielsystem ist der Faktor l gleich 0,1, also sinkt bei einer Programmlaufzeit von 16 Stunden die Effizienz bereits um etwa 10% ($T_0 = 2/\lambda$, vgl. Abbildung 3.3). Wird ein Duplikat der Sicherungspunkte angelegt, so ist l um etliche Größenordungen geringer und der Einfluß nicht mehr spürbar.

Die Sicherungszeit ist abhängig von der Auslastung der Systemresourcen und von der Vorgehensweise (transparent oder programmgesteuert). Durch nebenläufiges (asynchrones) Schreiben der Sicherungspunkte kann der Zugriff auf ein langsameres Speichermedium wie eine Festplatte beschleunigt werden. Allerdings wird freier Hauptspeicher benötigt, um die Sicherungsdaten puffern zu können. Je nachdem, wieviel Hauptspeicher die Applikation belegt, steht ein mehr oder weniger großer Pufferbereich zur Verfügung. Die Auslastung der Kommunikationsverbindungen bestimmt die zusätzliche Last, die durch das Versenden von Koordinierungsnachrichten beim Sichern entsteht. Da die Knoten mehr oder weniger gleichzeitig auf den Sicherungsspeicher zugreifen, beeinflußt die Anzahl der Knoten, die sich einen Sicherungsspeicher (z.B. eine Festplatte) teilen, die Sicherungsgeschwindigkeit, da die Zugriffe sequentialisiert werden müssen. Bei einem zentralen Server beispielsweise würde die Sicherungszeit linear mit der Prozessorzahl wachsen.

Im Vergleich ist die lokale Speicherung, z.B. auf einer privaten Festplatte, im Sinne der Geschwindigkeit vorteilhaft, allerdings für eine automatische Rekonfiguration nachteilig, da auf die Sicherungsdaten eines defekten Prozessors nicht mehr zugegriffen werden kann. In [DGH+93] wurde deshalb erstmals das Konzept „harter“ und „weicher“ Sicherungspunkte vorgestellt. Bei diesem Konzept werden zwei Arten von Sicherungspunkten erstellt: Harte Sicherungspunkte, auf die in jeder Situation zurückgesetzt werden kann, deren Speicherung aber aufwendiger ist und deshalb seltener durchgeführt wird, sowie weiche Sicherungspunkte, die zwar schnell gespeichert werden können, aber nicht in allen Fällen sicher sind, beispielsweise in einem Fall, in dem Rekonfiguration notwendig wird. Entsprechend könnten weiche Sicherungspunkte auf einer lokalen Platte angelegt werden, und harte Sicherungspunkte in einem größeren zeitlichen Abstand auf einem zentralen Server.

Für jeden Knoten liegen je nach Systemvoraussetzungen die Transferraten im Bereich von etwa 20 kB/s (zentraler Server) bis etwa 3 MB/s (lokale Festplatte). Falls die Sicherungsdaten komplett im Hauptspeicher gepuffert werden, kann eine Rate von bis zu 20 MB/s angesetzt werden. Für einen 10 MB großen Sicherungspunkt liegt die Speicherzeit im Bereich von 7 Minuten bis zu unter einer Sekunde (vgl. Abschnitt 5.6), hinzu kommt noch der Kommunikationsaufwand für die Koordinationsprotokolle, der je nach Verfahren und Verbindungsnetz ebenfalls unter einer Sekunde liegen kann (vgl. Abschnitt 6.3).

3.3.2 Das optimale Sicherungsintervall

Im Unterschied zu t_s ist das Sicherungsintervall t_i frei wählbar und kann bei Kenntnis der MTTF des Systems im Hinblick auf maximale Effizienz optimiert werden.

Damit eine Berechnung um einen Sicherungszyklus fortschreitet, kann unter der Annahme $t_r \approx 0$, $c \approx 1$ und $1/\mu \approx 0$ ein Zeitbedarf von

$$E[T] = \frac{1}{\lambda}(e^{\lambda(t_i+t_s)} - 1) \tag{3.10}$$

angesetzt werden. Die Überlegung, die zu dieser Formel führt, ist analog der Herleitung von Formel 3.3. Als Effizienz ergibt sich dann

$$E_{RFB}(t_i) = \frac{t_i\lambda}{e^{\lambda(t_i+t_s)} - 1} \tag{3.11}$$

In Abbildung 3.4 ist der Verlauf der Effizienzkurve eines Rechnersystems mit Rückwärtsfehlerbehebung in Abhängigkeit von der Länge des Intervalls zwischen zwei Sicherungspunkten aufgetragen. Die verschiedenen Kurven wurden für verschiedene Werte der Sicherungszeit t_s gewonnen. Alle Zeitangaben sind wiederum auf die MTTF des Systems normiert.

Ist das Sicherungspunktintervall kurz, so sinkt der Anteil des Nutzbetriebs, da mehr Zeit für die Rücksetzpunkterstellung aufgewendet wird. Wird das Intervall zu lang, so sinkt wiederum die Effizienz, da das Rechnersystem einen großen Teil der Rechenzeit für den Wiederholungsbetrieb nach einem Fehler aufwendet. Zwischen dem Zeitverlust durch das Rücksetzen im Fehlerfall und dem Verlust durch das mehr oder weniger häufige Speichern der Sicherungspunkte kann ein Optimum gefunden werden. Die Bestimmung dieses Optimums gelingt aber nur näherungsweise.

Im Verlauf eines Sicherungszyklus muß der Arbeitsabschnitt der Länge t_i bearbeitet werden, weiterhin muß ein Sicherungspunkt mit einem Zeitaufwand von t_s geschrieben werden. Unter den Annahmen $t_s \ll t_i$ (die Sicherungszeit ist klein im Vergleich zum Sicherungsintervall) und $1/\lambda \gg t_i$ (die MTTF ist groß gegenüber dem Sicherungsintervall) geht nach einem Fehler beim Zurücksetzen durchschnittlich ein halber Arbeitsabschnitt $\frac{t_i}{2}$ an Rechenfortschritt verloren. Während dem Sicherungsintervall sind $t_i\lambda$ Fehler zu erwarten. Auch während dem Rücksetzen können Fehler auftreten, allerdings kann dieser Einfluß unter der Annahme, daß nach einem Fehler das nochmalige Auftreten eines Fehlers im Wiederholungsbetrieb recht unwahrscheinlich ist ($\lambda^2 \ll \lambda$), vernachlässigt werden. Somit kann die folgende Näherung für Formel 3.10 (Zeitaufwand T in einem Sicherungszyklus) verwendet werden:

$$E[T] = t_i + t_s + \frac{t_i^2}{2}\lambda \tag{3.12}$$

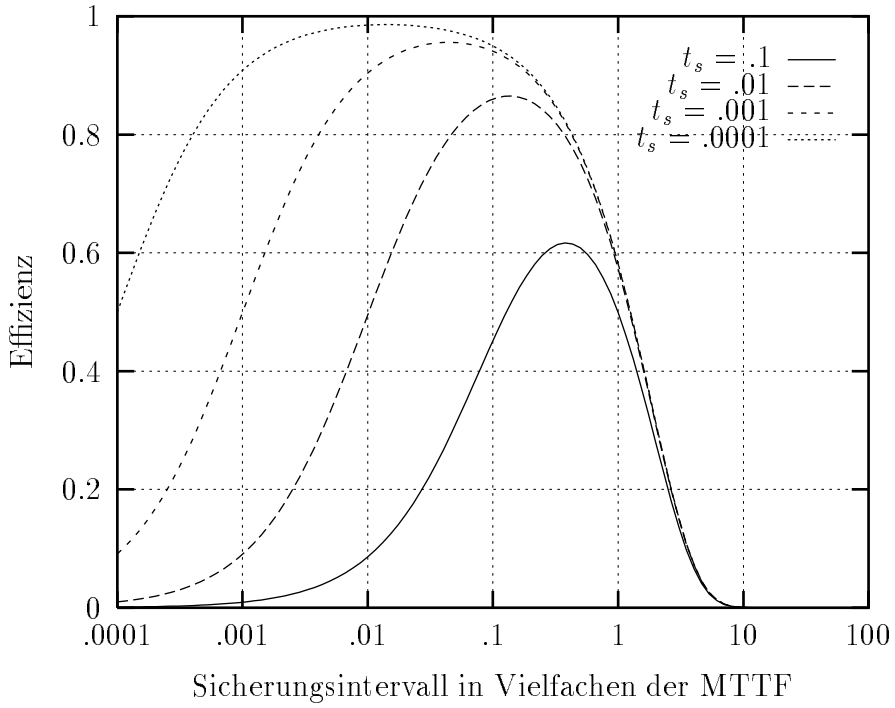

Abbildung 3.4: Effizienz eines Systems mit Rückwärtsfehlerbehebung

Der Bruch $\frac{t_i}{2}$ entspricht dem Zeitverlust t_v aus Abbildung 3.1. Die Erholungszeit t_r wurde zunächst vernachlässigt. Für die Effizienz ergibt sich dann die folgende Näherungsformel (vgl. Formel 3.11):

$$E_{RFB}(t_i) = \frac{1}{1 + \frac{t_s}{t_i} + \frac{t_i}{2}\lambda} \tag{3.13}$$

Diese Formel besitzt für

$$t_i = \sqrt{\frac{2t_s}{\lambda}} \tag{3.14}$$

ein Maximum (vgl. [You74]). Ähnliche Modelle mit z.T. anderen Voraussetzungen sind auch in [GD78, Dal79, Ech90] beschrieben.

Obwohl mit dieser Formel das Maximum bestimmbar ist, ist in der Realität im allgemeinen keine exakte Vorstellung von der Größe der MTTF eines Rechners gegeben. Wie Abbildung 3.4 zeigt, darf man sich, bei einer kurzen Sicherungszeit ($t_s \approx 0{,}0001 \cdot 1/\lambda$, durchaus auch einmal um eine Größenordung verschätzen, ohne allzu große Leistungseinbußen hinnehmen zu müssen, da für kleine Werte von t_s die Effizienz in der Umgebung des Optimums relativ flach verläuft.

Für das Zahlenbeispiel (MTTF $\approx$ 8 Stunden), liegt bei einer Sicherungszeit von etwa 32 Sekunden ($t_s = 0{,}001 \cdot$ MTTF) das optimale Sicherungsintervall bei etwa 20 Minuten (Effizienz etwa 95%).

3.3.3 Einfluß von fehlerbedingten Konsistenzproblemen

Besondere Fehlersituationen können auch durch mangelnde Koordination der Teilsysteme auftreten, beispielsweise wenn Fehler, die während der globalen Zustandssicherung erkannt werden, nicht verkraftet werden können. Eine mögliche Situation wäre der Fall, daß ein Fehler e auftritt, bevor Prozeß P_i seinen Sicherungspunkt erstellt, aber erst nachdem Prozeß P_j bereits einen neuen Sicherungspunkt gespeichert hat:

$$\exists P_{i,j} : e \rightarrow SP_i \wedge e \not\rightarrow SP_j$$

Nach der Modellvorstellung zu Abbildung 3.1 würde der Prozeß P_i den Sicherungspunkt SP_i nicht schreiben, sondern die Fehlerbehandlung beginnen. Der neue Sicherungspunkt von P_j ist zwar nicht durch den Fehler e beeinflußt, aber dennoch nicht konsistent zum aktuellen Sicherungspunkt von P_i. Solche Fehler verringern ebenfalls die Effizienz, da das Programm abgebrochen und wiederholt werden muß.

Bei gepuffertem Sichern dauert das physikalische Schreiben des Sicherungspunktes auf beispielsweise eine Festplatte erheblich länger als die durch t_s gegebene Zeit für das Sichern, da parallel zum Ablauf der Anwendung der Pufferspeicher entleert werden muß. Die Zeit vom Anstoßen des Schreibens bis zur Übertragung des letzten Datums auf das Sicherungsmedium sei als t_{ss} bezeichnet. Ist das System gegenüber Fehlern während dieser Zeit verletzlich, so tritt unter der Annahme $t_i + t_s \ll 1/\lambda$ ein nicht zu verkraftender Fehler mit der Wahrscheinlichkeit $\frac{t_{ss}}{t_i+t_s}$ auf. Die Rate der Fehler, die beim Sichern auftreten, ist entsprechend $\frac{t_{ss}}{t_i+t_s}\lambda$. Die Wahrscheinlichkeit $\frac{t_{ss}}{t_i+t_s}$ kann anstelle des Faktors l in die Formel 3.8 eingesetzt werden, so daß sich eine Rate von

$$\lambda_l = \frac{t_{ss}\lambda}{t_i + t_s} \tag{3.15}$$

für diesen Fehlerfall ergibt.

Bei einer Sicherungszeit t_s von 32 Sekunden (wie oben) und dem optimalen Sicherungsintervall von 20 Minuten wollen wir annehmen, das das asynchrone Sichern tatsächlich 1 Minute dauert. Bei einer Rechendauer des Programms von T_0 von 16 Stunden ergibt sich dann nochmals ein etwa 5%-iger Leistungsverlust.

Diese Verluste können durch geeignete Koordinierungsprotokolle, z.B. durch Zweiphasen-Freigabe, vermieden werden (siehe Kapitel 5). Der Zeitaufwand für diese Protokolle ist praktisch immer erheblich geringer als der potentielle Verlust durch Konsistenzfehler.

3.4 Fehlerbehebung

Der Aufwand zur Behebung eines Fehlers richtet sich nach seiner Auswirkung. Je nach Schwere des Fehlers genügt es, die betreffende Applikation zurückzusetzen, oder es ist ein Neustart des Systems notwendig, oder das Bedienungspersonal muß eingreifen um die Fehlerursache zu beseitigen. Manche Systeme sehen für diesen Fall eine automatische Rekonfigurationsmöglichkeit vor. Wegen der hohen Komplexität von Rekonfigurationsverfahren und wegen des zusätzlichen Hardwareaufwands stellt sich die Frage, ob sich für den gegebenen Anwendungsfall die dynamische Rekonfiguration überhaupt rentiert. Dazu ist zu klären, mit welcher Häufigkeit Fehler zu erwarten sind, die eine Rekonfiguration erfordern.

3.4.1 Fehlerauswirkungen und ihre Häufigkeit

Fehlerbedingte **Ausnahmen** (exceptions) haben Ursachen wie z.B. Speicherschutzverletzung, Paritätsfehler im Hauptspeicher oder Fehler in der Arithmetikeinheit. Erweiterte Hardware- bzw. Softwaremethoden zur Fehlererkennung (Kapitel 4) können ebenfalls Fehler durch eine Ausnahme anzeigen. Wenn der Fehler nur die Daten bzw. den Ablauf eines Prozesses der Applikation betrifft, so kann das Betriebssystem bei der Ausnahmebehandlung meistens den betroffenen Prozeß identifizieren: Fehler im nicht privilegierten Modus können unmittelbar einem Prozeß zugeordnet werden. Bei Betriebssystemen nach dem POSIX Standard [IEE90] wird der Prozeß mit einer Unterbrechung durch Signale wie SIGSEGV (Speicherschutzverletzung) oder SIGFPE (Fehler in der Arithmetikeinheit, z.B. Division durch Null) informiert. Signale führen üblicherweise zum Programmabbruch, können aber auch innerhalb des Programms bearbeitet werden.

Ausnahmen während eines Systemaufrufs (also im privilegierten Modus) sind häufig in ihrer Auswirkung begrenzt und können als **Fehlermeldung** an den aufrufenden Prozeß weitergegeben werden. Oft werden solche Ausnahmen durch das Übergeben falscher Parameter an den Systemaufruf ausgelöst.

Bei applikationsorientiertem Vorgehen können Ausnahmen und unerwartete Fehlermeldungen durch Rücksetzen der Anwendung behoben werden, systemorientiertes Vorgehen erfordert das Restaurieren des gesamten Systemzustands.

Andere Ausnahmen, die kein Signal auslösen und keine Fehlermeldung bewirken, führen zu einem **Systemabsturz**. Auch Softwarefehler im Betriebssystem (z.B. Verletzung von Konsistenzbedingungen in den Datenstrukturen des Betriebssystems) führen zum Systemabsturz, da eine exakte Bestimmung der Ursachen und Folgen eines Fehlers nicht möglich ist. Je nach Verflechtung der Rechnerknoten ist dann ein Neustart des Gesamtsystems oder nur eines einzelnen Knotens notwendig. Ist kein Sicherungspunkt des Betriebssystems vorhanden, so wird durch den Neustart

das System wieder in einen zulässigen Zustand gebracht (Vorwärtsbehebung). Anschließend können die Applikationen vom letzten Sicherungspunkt ausgehend neu gestartet werden.

Im Gegensatz zu den bisher genannten Fehlerauswirkungen, die vorwiegend für temporäre Fehlern auftreten, ist bei einem **Totalausfall** eines Rechnerknotens der Neustart des betroffenen Prozessors nicht möglich. Ursachen sind z.B. Stromausfall oder dauerhafte Hardwarefehler. Bei dieser Art von Fehlern ist Rekonfiguration (Ausgrenzen) oder die Reparatur des defekten Knotens erforderlich. Nach der Beseitigung des Fehlers stehen Sicherungspunkte zur Verfügung, von denen aus die Applikation wie beim Neustart wieder angefahren werden kann. Auch bei Verbindungsausfällen oder gar einer Netzwerkpartitionierung kann eine Rekonfiguration oder Reparatur notwendig werden.

Durch **Nachrichtenverlust** können einzelne oder, im Falle eines dauerhaften Verbindungsausfalls, mehrere Nachrichten verloren werden. Fehler dieser Art können durch geeignete Verbindungsprotokolle erkannt und teilweise maskiert werden. Weniger aufwendige Verfahren verwenden einfache Zeitschrankenüberwachung. Oft können Nachrichtenverluste auch auf Fehler in einem der Endpunkte zurückgeführt werden und dort eine Ausnahme oder Fehlermeldung auslösen. Auch Nachrichten, die durch einen Fehler verfälscht werden, kann man zu den verlorenen Nachrichten zählen, da solche Fehler durch Prüfsummenbildung mit hoher Wahrscheinlichkeit erkannt werden.

Untersuchungen von [Sie90, IV85] lassen erkennen, daß vorübergehende Hardwarefehler etwa um ein bis zwei Größenordnungen häufiger als dauerhafte Fehler zu erwarten sind. Die folgende Tabelle versucht, diese Fehler mit der Schwere ihrer Auswirkung zu korrelieren. Die für die Behebung notwendige Maßnahme ist in der linken Spalte aufgezählt. Softwarefehler des Betriebssystems wurden nicht in die Betrachtungen einbezogen, obwohl sie in beiden Literaturbeispielen etwa genauso häufig wie Hardwarefehler auftreten. Die Anzahl und Wirkung dieser Fehler ist erheblich abhängig von der Stabilität, Reife und Komplexität des Betriebssystems. Es sei an dieser Stelle nur angenommen, daß Betriebssystemfehler sich ähnlich wie transiente Hardwarefehler verhalten und keinen maßgeblichen Einfluß auf die Fehlerverteilung haben. Wie in Abschnitt 2.3 bereits gesagt, werden Softwarefehler einer Applikation nicht betrachtet.

Behebungsmaßnahme	HW-perm.	HW-trans.	Summe
Rücksetzen	–	90%	90%
Neustart	–	5%	5%
Rekonf./Reparatur	5%	–	5%
Fehler gesamt	5%	95%	100%

Für obige Tabelle wurde von einem dauerhaften Fehler je 20 Fehler ausgegangen. Dauerhafte Fehler (5% aller Fehler) erfordern grundsätzlich Reparatur oder Re-

konfiguration. Da sich der Rechner die meiste Zeit mit „sinnvollen“ Dingen und nicht mit Betriebssystemaufgaben beschäftigt, wird der Löwenanteil (über 90%) der transienten Hardwarefehler auf die Applikation einwirken und kann durch Rücksetzen behoben werden. Etwa 5% der transienten Fehler, die das Betriebssystem beeinflussen, führen zu einem Neustart des betroffenen Knotens bzw. des gesamten Multiprozessors.

3.4.2 Abschätzung der Rücksetzzeit

Die Rücksetzzeit t_r setzt sich zusammen aus einer Reaktionszeit, bis das System die Fehlerbehebung einleitet, und einer Erholungszeit, bis eine Applikation wieder fortgesetzt werden kann. Die Reaktionszeit wird von der Fehlerlatenz und damit von der Art des Fehlers bestimmt. Durch Selbsttests erkannte Fehler können praktisch sofort das Einleiten der Fehlerbehebung anstoßen. Da Totalausfälle vorwiegend durch Fremdtests erkannt werden, ist die Verzögerung durch die Latenz von Fremdtests für diese Fehlerart zur Rücksetzzeit t_r zu addieren. Die Latenz von Fremdtests ist zwar wegen der notwendigen Kommunikation im Vergleich zur Latenz von Selbsttests hoch, allerdings können sich viele Fehler, die durch Fremdtests erkannt werden (Systemabsturz, Stromausfall), nicht mehr ausbreiten.

Die Erholungszeit ist von der Schwere der Fehlerbehebungsmaßnahme abhängig. Für Fehler die mittels Rücksetzen behoben werden können, entspricht die Erholungszeit in etwa der Sicherungszeit, aber das Rücksetzen auf einen Systemsicherungspunkt oder ein Neustart ist aufwendiger als das Rücksetzen einer einzelnen Applikation.

Am aufwendigsten sind Fehler die Reparatur oder Rekonfiguration erfordern. Reparatur erfordert das Eingreifen des Bedienungspersonals oder eines Servicetechnikers. Zur Rekonfiguration, z.B. bei einem degradierenden System oder bei Einsatz von Reserveprozessoren, sind über die einfache Fehlererkennung hinausgehende weitere Diagnosealgorithmen notwendig, die Fehler für mehr oder weniger beliebige Fehlermuster exakt lokalisieren. In [GD94, Alt93] befindet sich eine ausführliche Übersicht über Diagnoseverfahren und ihre Eignung für Multiprozessoren.

Bei der Rekonfiguration mit Reserveprozessoren ergeben sich Abhängigkeiten von der Topologie des betrachteten Rechners und von der Topologie der Kommunikation innerhalb einer Anwendung. Anwendungen mit intensiver Kommunikation sind oft für eine bestimmte Rechnertopologie optimiert, so daß bei einem dauerhaften Ausfall bestimmte Nachbarschaften beachtet werden müssen, um erhebliche Leistungsverluste zu vermeiden. Das Auffinden einer geeigneten neuen Abbildung der Applikation auf den fehlerhaften Multiprozessor ist NP-vollständig. Untersuchungen dazu wurden am DIRMU-System durchgeführt [Mor86]. In einem degradierenden System ist Lastumverteilung notwendig, um die Aufgabe möglichst gleichmäßig auf die verbleibenden Prozessoren aufzuteilen. Änderungen im Systemzustand müssen von der

Ebene des Applikationsprogramms aus mit einem erheblichen Implementierungsaufwand behandelt werden. Transparente Verfahren sind nicht ohne weiteres möglich.

Das Einlesen von Sicherungspunkten bedeutet etwa den gleichen Aufwand wie das Schreiben. Für das einfache Rücksetzen einer Applikation muß beim derzeitigen Stand der Technik eine Zeit von etwa 1–7 Minuten angerechnet werden (vgl. Abschnitt 3.3.1, Datenübertragungszeit) wobei nicht von der Pufferung profitiert werden kann. Für einen Neustart ist nochmals eine Zeit von etwa 5 Minuten hinzuzählen. Der Eingriff des Bedienpersonals (Reparatur) kann z.B. mit durchschnittlich zwei Stunden angesetzt werden. Für ein System mit Rekonfiguration sei angenommen, daß die Rekonfiguration etwa den gleichen Zeitbedarf wie ein Neustart hat. Die mittlere Rücksetzzeit liegt mit diesen Annahmen und unter Berücksichtigung der Fehlerverteilung aus Abschnitt 3.4.1 bei etwa 7–14 Minuten (Reparatur durch das Bedienungspersonal), oder bei etwa 2–9 Minuten (automatische Rekonfiguration). Bei automatischer Rekonfiguration müssen eher größere Ladezeiten angesetzt werden, da die Sicherungspunkte zum Wiederherstellen der Applikation auf einem allgemein zugänglichen Server abgespeichert werden, während ohne Rekonfiguration von der schnellen, lokalen Speicherung profitiert werden kann. Dies deutet darauf hin, daß Rekonfiguration vermutlich nur bei einem hierarchischen Sicherungskonzept mit harten und weichen Sicherungspunkten (vgl. Abschnitt 3.3.1) sinnvoll sein kann.

Auch die Rücksetzzeit wirkt sich auf die Effizienz des Systems aus. Pro Fehler geht zusätzlich etwa eine Zeit von t_r verloren. Während eines Sicherungsintervalls t_i ist also näherungsweise ein Verlust von $t_r t_i \lambda$ zu erwarten und kann in Formel 3.12 angesetzt werden. Formel 3.13 verändert sich entsprechend zu:

$$E_{RFB}(t_i) = \frac{1}{1 + \frac{t_s}{t_i} + \frac{t_i}{2}\lambda + t_r\lambda} \tag{3.16}$$

Die Rücksetzzeit wirkt sich nicht auf die Lage des optimalen Sicherungspunktintervalls aus, sondern nur auf die Höhe des Maximums, weil der zusätzliche Term im Nenner nicht von t_i abhängt und daher bei der Differentiation entfällt.

Unter Verwendung derselben Zahlen wie bisher (MTTF $\approx$ 8 Stunden) geht bei einer Rücksetzzeit t_r von 10 Minuten die maximale Effizienz nochmals um etwa 2% zurück.

3.5 Schlußfolgerungen

Die Effizienz eines Parallelrechners wird bestimmt von der Sicherungszeit, von der Rücksetzzeit, und von der Fehlerlatenz. Über die Fehlerlatenz ergibt sich auch eine Abhängigkeit von der Länge eines Rechenauftrags. Abbildung 3.5 soll einen Anhaltspunkt geben, für welche Zeitwerte dieser Größen die Effizienz deutlich beeinflußt wird. Dargestellt sind Schwellwerte, für die, falls sie überschritten werden, der

entsprechende Einfluß mehr als 5% der Rechenleistung vernichtet. Bei einem Parallelrechner mit 1000 Prozessoren (MTTF eines Einzelrechners 1 Jahr) beispielsweise kann bei einer Sicherungszeit von 30 Sekunden nur noch eine maximale Effizienz von 95% erreicht werden, dauert das Sichern länger, so sinkt die Effizienz unter 95%. Zur Berechnung der Kurve für die maximale Sicherungszeit wurde das optimale Sicherungsintervall (siehe Abschnitt 3.3.2) angesetzt.

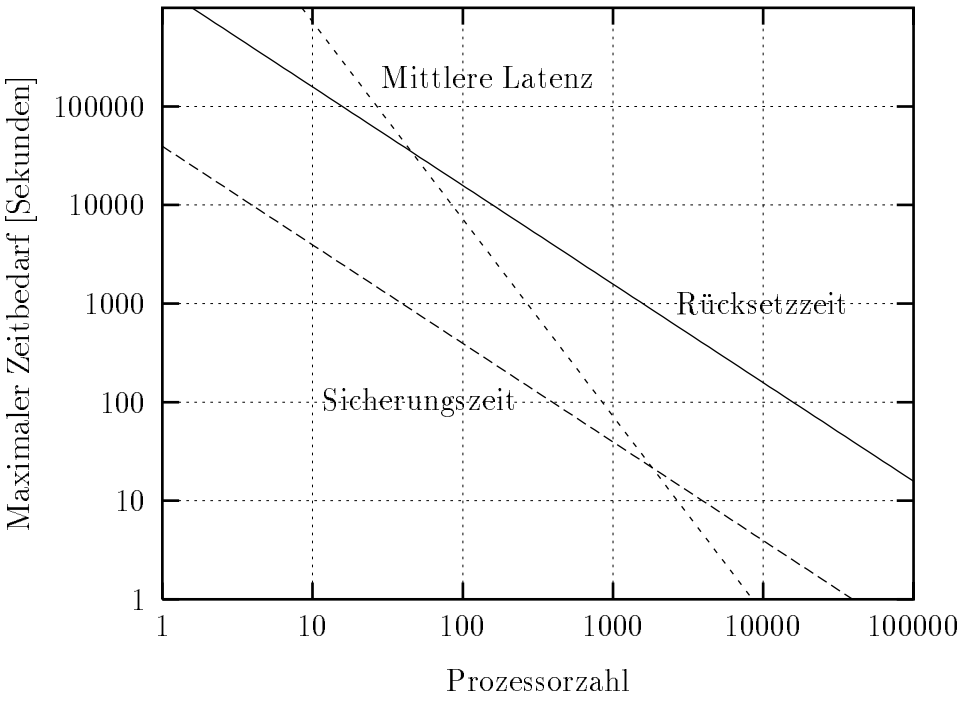

Abbildung 3.5: Anforderungen an Sicherungs-, Rücksetz- und Fehlerlatenzzeit

Die Zeitreserve der Sicherungszeit ist bereits ab etwa 100 Prozessoren gering, d.h. der Einsatz sehr schneller Datenpfade und sehr effizienter Sicherungsverfahren wirkt sich günstig auf die Effizienz des Rechners aus. Bei mehr als 10000 Prozessoren ist mit derzeitiger Technologie noch keine gute Effizienz erreichbar. Aufgrund der im interessanten Bereich deutlichen Vorteile schnellerer Zustandssicherung ist der applikationsorientierte bzw. programmgesteuerte Ansatz dem transparenten vorzuziehen.

Der Einfluß der Rücksetzzeit ist um ein bis zwei Größenordnungen kleiner und beginnt entsprechend erst ab einer Prozessorzahl > 10000 (5%-Grenzwert der Rücksetzzeit bei etwa zweieinhalb Minuten) zu wirken. Automatische Rekonfiguration kann sich also nur jenseits dieser Grenze, wenn überhaupt, günstig auswirken.

Der Einfluß der Fehlerlatenz hängt von der Länge des Sicherungszyklus ab, sowie außerdem von der Länge der Berechnung. Für t_i und t_s wurden dieselben Werte wie für die Kurve der Sicherungszeit verwendet, d.h. ausgehend von einer Effizienz

von 95% und einer Fehlerrate λ wurde die zugehörige Sicherungszeit und das entsprechende optimale Sicherungsintervall bestimmt. Als Rechenzeit der Applikation T_0 wurden unabhängig von der Prozessorzahl 24 Stunden angesetzt. Dadurch, daß die Sicherungszyklen mit zunehmender Prozessorzahl immer kürzer werden müssen, und weil die Fehlerrate mit zunehmender Prozessorzahl steigt, fällt die Kurve der maximal erlaubten mittleren Fehlerlatenzzeit steiler als die anderen beiden Kurven. Bei einer Berechnung, die auf einem Parallelrechner mit 1000 Prozessoren 24 Stunden erfordert, sollte also die mittlere Fehlerlatenz im Bereich unter einer Minute sein. Mit konkurrenten Erkennungsmaßnahmen kann dieser Eckwert leicht erreicht werden. Bei etwa 10000 Prozessoren sinkt die maximal erlaubte Fehlerlatenz unter eine Sekunde.

Die Zuverlässigkeit des Sicherungsspeichers beeinflußt ebenfalls die Effizienz. Da die Zuverlässigkeit durch das Verwenden von RAID-Systemen oder gespiegelten Festplatten über einen weiten Bereich an die Bedürfnisse angepaßt und skaliert werden kann, wurde dieser Einfluß in Abbildung 3.5 nicht nochmals hervorgehoben.

Die maximale Rechenzeit einer Applikation wird durch die Fehlerüberdeckung beschränkt. Ohne erweiterte Fehlererkennungsmaßnahmen ($c \approx 80\%$) sollte für allgemeine Applikationen eine maximale Rechenzeit von etwa 5000 Prozessorstunden nicht überschritten werden. Für aufwendige Berechnungen ist daher eine deutliche Steigerung der Fehlerüberdeckung zwingend. Bei weniger fehleranfälligen Algorithmen ist dieser Einfluß nicht in einem so hohen Maß entscheidend.

Überschlägig können aufgrund der in diesem Kapitel gewonnenen Erkenntnisse folgende Anforderungen für den Entwurf von massiv parallelen, skalierbaren Rechnersystemen (mehr als 10000 Prozessoren) angegeben werden:

- Fehlerüberdeckung größer als 99,5%,
- mittlere Fehlerlatenz kleiner als eine Sekunde,
- Mehraufwand zum Speichern der Sicherungspunkte im Bereich weniger Sekunden,
- Rücksetzzeit für transiente Fehler unter einer Minute,
- maximal 1 Fehler des Sicherungsspeichers je 10^{15} Bits.

Damit haben sich als die großen Herausforderungen für den Entwurf von massiv parallelen Rechnersystemen zwei Themengebiete herauskristallisiert:

(1) Der Entwurf von Fehlererkennungsmechanismen mit hoher Fehlerüberdeckung und geringer Latenz bei geringem Mehraufwand, sowie

(2) der Entwurf schneller Zustandssicherungsverfahren.

Softwareverfahren zu diesen beiden Themen werden in Kapitel 4 und 5 ausführlicher behandelt. Grundsätzlich sollten aber auch die Einzelrechner mit größter Sorgfalt entworfen werden, um von vornherein eine möglichst geringe Fehlerrate zu erzielen. Die Systemsoftware muß solide getestet und soweit als möglich fehlerfrei sein. Im Zweifelsfall sind immer einfache und überschaubare Lösungen vorzuziehen.

Als nicht notwendig in diesem Anwendungsbereich erweisen sich Verfahren zur dynamischen Rekonfiguration. Die Umsetzung der Rückwärtsfehlerbehebung für numerische Parallelrechner kann sich darauf beschränken, nur vorübergehende Fehler automatisch zu beheben. Für die restlichen 5% der Fehler kann die Fehlerbehebung im „Handbetrieb“ günstiger vorgenommen werden. Diese Überlegung basiert leider nur auf Erfahrungswerten anstatt auf Fakten (vgl. Abschnitt 3.4.1). Der möglicherweise maßgebliche Einfluß der Betriebssystemfehler auf die Fehlerverteilung wurde völlig außer Acht gelassen. Allerdings kann man mit an Sicherheit grenzender Wahrscheinlichkeit davon ausgehen, daß Betriebssystemfehler nicht die Häufigkeit von Fehlern erhöhen, deren Behebung eine Rekonfiguration erfordert.

Kapitel 4

Kontrollflußselbstüberwachung

Im vorherigen Kapitel hat sich gezeigt, daß eine hohe Fehlerüberdeckung Grundvoraussetzung für den Bau von massiv parallelen Rechnern ist. Übliche Methoden sind nicht ausreichend, um eine Fehlerüberdeckung größer als 90% zu erreichen. Insbesondere zum Erkennen von CPU-Fehlern sind zusätzliche Maßnahmen mit möglichst sparsamem Redundanzeinsatz wünschenswert.

Basierend auf Kontrollflußüberwachung wird hier ein neuartiges Softwareverfahren vorgestellt, die „Kontrollflußselbstüberwachung". Bei ähnlichen Fehlererkennungseigenschaften wie bei der Kontrollflußüberwachung durch einen Watchdogprozessor ist weder eine applikationsspezifische Anpassung, noch zusätzliche Hardware nötig. Dieser Ansatz ist daher eine kostengünstige Alternative zum Einsatz eines Watchdogprozessors. Die Methode ist unabhängig von der Systemhardware und ermöglicht daher ihren Einsatz in einer Vielzahl von Systemen ohne großen Portierungsaufwand.

4.1 Kontrollflußüberwachung

Viele Hardwarefehler manifestieren sich durch Störungen des normalen Programmlaufs, d.h. durch Abweichungen des Programmflusses von der Spezifikation. Nach einer Untersuchung in [Mic92], basierend auf Fehlerinjektionsversuchen und Beobachtungen in [STDM82, KGT89, CS90] ist allein für die Kontrollflußüberwachung eine relativ hohe Fehlerüberdeckung von etwa 60–70% aller CPU-Fehler zu erwarten. Entsprechend wurden Methoden der Kontrollflußüberwachung entworfen, die unter Verwendung eines relativ einfachen Koprozessors, einem sogenannten „Watchdogprozessor", das Verhalten eines Programms zur Laufzeit beobachten.

Bei Überwachungsverfahren, die auf der „derived signature analysis" basieren, wird Information über den Programmfluß in Signaturen komprimiert und in den Programmcode eingebettet. Zur Laufzeit überwacht der Watchdogprozessor die Adreß-

und Datenleitungen des Prozessors. Er erhält dabei neben der aktuellen Laufzeitinformation auch die Referenzinformation um den Programmablauf zu verifizieren [ST82, SS83, ES84, SS87, Sos88, WS90b, WS90a, SS91]. Ist die CPU mit einem internen Cache ausgestattet, so muß der Watchdogprozessor in den Prozessorbaustein selbst integriert werden, da sonst der Instruktionsstrom nicht mehr beobachtbar ist.

Beim anderen Ansatz zur Überwachung („assigned signature analysis“) existiert dieses Problem nicht. Nicht jede Instruktion wird überwacht, sondern der Watchdogprozessor prüft nur die korrekte Reihenfolge von Signaturen, die explizit vom Prozessor zum Watchdogprozessor übertragen werden. Allerdings bleiben kurze Fehlsprünge (z.B. wenn ein Befehl nicht ausgeführt wird) unerkannt. Entsprechend ist die Fehlerüberdeckung dieses Ansatzes geringer.

Die Kontrollflußselbstüberwachung basiert auf dem letzteren Ansatz. Ausgangsposition ist das bereits in [Lu82] vorgestellte „SIC“-Verfahren (Structural Integrity Checking). Erweiterungen und Verbesserungen wurden in [MH91, Mic92] (Extended-SIC) und [PMHH93] (SEIS – Signature Encoded Instruction Stream) vorgestellt. Im SIC-Verfahren ist der Watchdogprozessor ein eigenständiger Rechner. Die Referenzinformation über den Programmablauf des Hauptprogramms ist ein ausführbares Programm für den Watchdogprozessor.

Im ESIC-Verfahren wird durch den Watchdogprozessor das Verhalten eines deterministischen Kellerautomaten nachgebildet. Die Referenzinformation wird vor dem Start in Form einer Automatentafel dem Watchdogprozessor übergeben. Im SEIS-Verfahren schließlich ist der Watchdogprozessor nur noch ein Automat vergleichsweise geringer Komplexität, im wesentlichen ein Mehrfachkomparator. Die Laufzeitinformation wird ebenfalls explizit zum Watchdogprozessor übertragen, aber jede Signatur enthält bereits Information über mögliche Folgesignaturen. Gesonderte Referenzinformation entfällt bei diesem Ansatz.

Zum Übertragen der Laufzeitsignaturen muß bei allen drei Verfahren das Originalprogramm verändert werden. Ein Präprozessor analysiert zunächst den Programmtext auf Hochsprachenebene und erstellt daraus einen Programmflußgraphen (CFG). Jeder Knoten im Programmflußgraph entspricht einer verzweigungsfreien Folge von Anweisungen, jede Kante dem Programmfluß zwischen den Anweisungsfolgen. Im Anschluß an die Programmflußanalyse wird der modifizierte Programmtext erzeugt, der Befehle zum Übertragen der Signaturen zum Watchdogprozessor enthält. Im Falle des SIC-Verfahrens wird außerdem das Watchdog-Programm erzeugt, für das Beispiel des ESIC-Verfahrens (siehe Abbildung 4.1) erzeugt der Präprozessor eine tabellarische Wiedergabe des Programmflußgraphen, die Watchdog-Referenz, die vom ESIC-Watchdogprozessor verwendet wird, um das Laufzeitverhalten des Programms zu kontrollieren. Falls ein Ablauf erkannt wird, der nicht durch eine Kante im Programmflußgraphen abgedeckt ist, so löst der Watchdogprozessor eine Ausnahme aus.

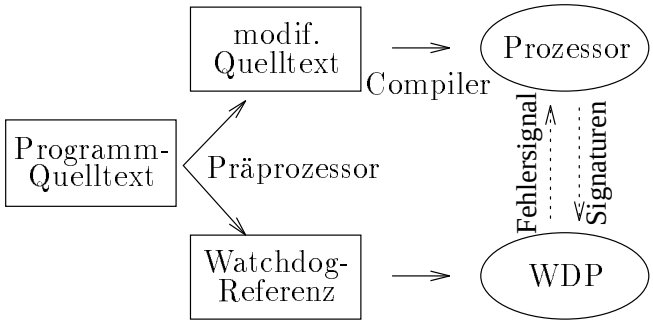

Abbildung 4.1: Das ESIC-Verfahren

Im Gegensatz zu der in [Lu82] beschriebenen SIC-Methode, ist das ESIC-Verfahren in der Lage, Unterbrechungen und Prozedurvariablen korrekt zu behandeln. Viele Programmiersprachen erlauben es, daß Funktionen oder Prozeduren über Zeiger aufgerufen werden, so daß der mögliche Programmfluß nicht allein durch Syntaxanalyse im Präprozessor bestimmt werden kann.

Auch das Auftreten von Unterbrechungen kann nicht statisch vorhergesagt werden. Ein Sprung zu einer Interrupt-Behandlungsroutine kann zu einem beliebigen Zeitpunkt auftreten. In ESIC wird deshalb nicht der gesamte Programmflußgraph aufgestellt, sondern nur eine Menge von Subgraphen, die jeweils nur den Ablauf in einer Prozedur- bzw. Funktion darstellen. Es entsteht also eine Menge von Subgraphen CFG^f, wobei f eine Nummer ist, die die zugehörige Funktion eindeutig identifiziert. Kontrollflußübergänge durch Funktionsaufrufe werden nicht im Programmflußgraphen berücksichtigt.

Die Signaturen, die zum Watchdogprozessor übertragen werden, bestehen aus der Knotennummer im Programmflußgraph und der Funktionsnummer f. Der Watchdogprozessor überprüft nicht nur, ob die Knotennummer innerhalb von CFG^f korrekt ist, sondern er überwacht auch die Funktionsnummer. Die Nummer darf sich nur ändern, wenn eine Funktion aufgerufen wird, wobei der Kontrollfluß zu einem anderen Subgraphen übergeht, beginnend mit dem Einsprungpunkt der Funktion. Alle anderen Übergänge werden als ein Fehler angezeigt. Wird eine Funktion betreten, so legt der Watchdogprozessor die vorherige Signatur und Funktionsnummer auf seinem Stack ab. Um die Überwachung innerhalb des aufrufenden Programms wieder fortzusetzen, werden beim Verlassen eines Unterprogramms die Funktionsnummer und Signatur wieder vom Stack geladen. Ein Nachteil dieser Methode ist es, daß die Latenz in manchen Fällen sehr hoch sein kann, denn falls das Programm rekursive Funktionsaufrufe enthält, werden Fehler möglicherweise erst nach der Rückkehr vom letzten Unterprogrammaufruf erkannt. Der Aufruf einer falschen Funktion anstelle einer anderen Funktion kann ebenfalls nicht erkannt werden. Fehler dieser Art sind aber sehr unwahrscheinlich.

4.2 Das Selbstüberwachungsverfahren

Die Kontrollflußselbstüberwachung basiert auf der ESIC-Methode. Das Programm wird ebenfalls durch einen Präprozessor zu einem neuen, selbstüberwachenden Programm umgewandelt. Die Referenzinformation über den Programmlauf ist aber bereits im Programm enthalten und muß nicht mehr gesondert generiert werden (vgl. Abbildung 4.2). Bei der Überwachung muß keine Signatur zu einer externen Einheit übertragen werden. Der Programmcode zur Selbstüberwachung kann also vom internen Cache eines Prozessors profitieren.

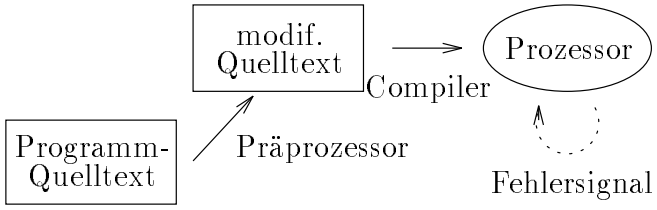

Abbildung 4.2: Das Selbstüberwachungsverfahren

Ein Nachteil des Verfahrens gegenüber dem Verwenden eines Watchdogprozessors ist, daß die überwachte Einheit und der Überwacher identisch sind. Da die Überwachungsmethode ebenfalls auf der „assigned signature analysis“ basiert, waren bei Fehlerüberdeckung und Fehlerlatenz dennoch ähnliche Ergebnisse zu erwarten wie für einen vergleichbaren Watchdogprozessor, denn die Ablaufsteuerung des Prozessors und die Überprüfung des Ablaufs durch Selbstüberwachung sind grundsätzlich verschieden. Ein Fehler, der den Kontrollfluß verändert und gleichzeitig die Selbstüberwachung so beeinflußt, daß er nicht erkannt wird, ist unwahrscheinlich.

Eine ähnliche Methode zur Kontrollflußüberwachung wurde auch in [YC80] vorgestellt. Das BSSC-Verfahren (Block Signature Self Checking) aus [MKGT92] verfolgt ebenfalls einen ähnlichen Ansatz, der auf Assembler-Ebene arbeitet. Keine der anderen Methoden ist aber in der Lage, Interrupts und Prozedurvariablen korrekt zu bearbeiten.

4.2.1 Kontrollflußanalyse

Die Kontrollflußanalyse des Präprozessors wurde mit Hilfe des `bison` Parser-Generators implementiert. Aus einer Beschreibung der Kontrollstrukturen der Programmiersprache „C“ in BNF wird ein Programm erzeugt, daß diese Strukturen erkennen kann. Die Programmflußgraphen CFG^f werden aus Grundbausteinen, die einfache Anweisung, Schleifen oder bedingte Sprünge darstellen, aufgebaut. Jeweils an

den Knoten des Graphen wird ein Stück Programmcode zur Sebstüberwachung eingefügt.

Die Knoten der Programmflußgraphen beginnen jeweils zu Anfang eines verzweigungsfreien Intervalls. Zusätzlich wird nach jeder SF-ten Hochsprachenanweisung ein weiterer Knoten eingefügt, um die Überwachung in annähernd regelmäßigen Abständen durchzuführen. Die maximale Anzahl von Hochsprachenanweisungen, die durch einen Knoten repräsentiert werden darf, wird durch den Parameter SF („Statement-Faktor") bestimmt. Kleine Werte für SF erhöhen die Fehlerüberdeckung und verringern die Fehlerlatenz. Größere Abstände verringern den zusätzlichen Laufzeit- und Speicherbedarf, der durch die Überwachung entsteht.

In Programmen mit nur kurzen verzweigungsfreien Intervallen entsteht nach diesem Verfahren auch bei großen Statement-Faktoren eine hohe Knotendichte. Deshalb können mit Hilfe eines Reduktionsverfahrens auch Knoten innerhalb von kurzen Verzweigungen entfernt werden. Für dieses Verfahren besitzen alle Knoten ein Gewicht, das sich aus der Anzahl der Hochsprachenanweisungen errechnet, die sie repräsentieren. Knoten eines Programmflußgraphen dürfen nur dann reduziert werden, wenn das Gewicht der dabei neu entstehenden Knoten den vorgegebenen Statement-Faktor nicht überschreitet. Weiterhin dürfen Zyklen im Programmflußgraphen nicht entfernt werden, um sicherzustellen, daß jede Programmschleife mindestens einmal den Selbstüberwachungscode enthält, da sonst das Programm sehr lange ohne Überwachung laufen könnte. Die Programmanalyse und das Reduktionsverfahren werden in [Hön90, Mic92] genauer beschrieben.

4.2.2 Überwachung des Programmablaufs

Wie im ESIC-Verfahren werden bei der Selbstüberwachung ebenfalls die folgenden drei Typen von Knoten des Programmflußgraphen unterschieden:

Start of function (SOF). Der Einsprungpunkt einer Funktion ist als SOF-Knoten markiert. SOF-Knoten haben keine einlaufenden Kanten.

End of function (EOF). Diese Knoten markieren das Ende einer Funktion oder eine `return`-Anweisung. EOF-Knoten haben keine auslaufenden Kanten.

Normaler Knoten. Alle anderen Knoten sind normale Knoten.

Anhand der SOF-Markierung kann festgestellt werden, ob ein gültiger Funktionsaufruf stattgefunden hat. Anhand der EOF-Markierung wird erkannt, daß die aktuelle Funktion verlassen wird. Abbildung 4.3 zeigt ein kleines Programmfragment und den zugehörigen Kontrollflußgraphen. Die Einstellung war $SF = 1$ ohne Reduktion.

Die Funktion `example()` darf nur bei v_0 (SOF) betreten werden. Im Beispiel ist ein gültiger Nachfolger des Knotens v_0 sowohl der Knoten v_1 bzw. v_4, je nachdem, ob

Abbildung 4.3: Beispiel eines Programmflußgraphen

die `while`-Schleife betreten wird oder nicht. Der Endknoten, an dem die Funktion verlassen wird (EOF) ist der Knoten v_4.

Das Überwachungsverfahren wurde mit dem Ziel eines möglichst geringen Laufzeitmehraufwands entworfen. Um die Kontrollflußselbstüberwachung zu beschleunigen, sind dazu die einzelnen Subgraphen der Funktionen $CFG^f = (V^f, E^f)$ als Adjazenzmatrizen („Kontrollflußmatrizen“ CFM^f) repräsentiert. Die Elemente cfm_{ij}^f einer solchen Matrix sind wie folgt definiert:

$$\forall v_i^f, v_j^f \in V^f \wedge (v_i^f, v_j^f) \in E^f : \quad cfm_{i,j}^f = f$$
$$\forall v_i^f, v_j^f \in V^f \wedge (v_i^f, v_j^f) \notin E^f : \quad cfm_{i,j}^f = 0$$

Der erste Knoten eines CFG (SOF) erhält dabei immer den Index 0. Der Überwachungscode greift immer auf die Kontrollflußmatrix der gerade laufenden Funktion f zu. Falls in einem normalen Knoten v_j^f der vorherige Knoten v_i^f ein gültiger Vorgänger war, befindet sich eine entsprechende Kante in E^f. Also wird der Zugriff auf die Kontrollflußmatrix mit den Indizes i und j an der Stelle $cfm_{i,j}^f$ den Wert f zurückliefern. Falls der ausgelesene Wert gleich Null ist, hat ein illegaler Ablauf stattgefunden. Ist der ausgelesene Wert weder 0 noch f, so ist ein illegaler Übergang von einer anderen Funktion aus aufgetreten.

Für das Beispiel (Abbildung 4.3) ergibt sich folgende Kontrollflußmatrix:

$$CFM^f = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ f & 0 & 0 & f & 0 \\ 0 & f & 0 & 0 & 0 \\ 0 & f & f & 0 & 0 \\ f & 0 & 0 & f & 0 \end{pmatrix}$$

Da der Startknoten der Funktion keine einlaufenden Kanten besitzt, ist die erste Zeile der Kontrollflußmatrix nur mit Nullen besetzt. Ebenso steht für jeden Endknoten (im Beispiel für v_4) eine mit Null besetzte Spalte in der Matrix, da EOF-Knoten keine auslaufenden Kanten besitzen.

Die Darstellung der Programmablaufreferenz ist für die möglichst schnelle Ausführung der Überwachungscodesequenzen ausgelegt. Nachteilig ist bei dieser Art der Speicherung, daß der Speicherbedarf der Kontrollflußmatrix quadratisch mit der Zahl der Knoten innerhalb der einzelnen Funktionen wächst. Da außerdem Programme vorwiegend sequentiell ablaufen, ist in der Kontrollflußmatrix vorwiegend nur die Umgebung der Hauptdiagonale besetzt. Eine kompaktere Speicherung wäre durchaus möglich, ist aber im Sinne der Geschwindigkeit nicht wünschenswert. Für strukturierte, modulare Programme mit nur kurzen Funktionsdefinitionen wird sich auch der quadratische Einfluß auf den Speichermehraufwand nur in einem geringen Maß auswirken.

Je nach Typ eines Knotens fügt der Präprozessor verschiedene Programmstücke zur Selbstüberwachung in das Quellprogramm ein. In diesen Programmstücken wird grundsätzlich nur die Funktionsnummer der aktuell laufenden Funktion in einer globalen Variable gespeichert, alle anderen Variablen des Überwachungsverfahrens sind lokal und bleiben während eines Funktionsaufrufs auf dem Stack erhalten. Beim Verlassen einer Funktion werden sie automatisch deallokiert. Explizite Stackoperationen wie im ESIC-Verfahren sind also nicht mehr notwendig.

- Bei Betreten einer Funktion f (SOF-Knoten) wird die Nummer der aufrufenden Funktion lokal zwischengespeichert. Die globale Variable, die f enthält, wird aktualisiert. Funktionen dürfen jederzeit aufgerufen werden, also wird die Signaturfolge nicht überprüft.

- Bei einem normalen Knoten wird der Programmfluß wie oben beschrieben durch das Auslesen eines Elements der Kontrollflußmatrix verifiziert.

- Wird ein EOF-Knoten erreicht, so wird ebenfalls der Programmfluß durch Auslesen des entsprechenden Elements der Kontrollflußmatrix verifiziert. Zusätzlich wird die Funktionsnummer der aufrufenden Funktion wiederhergestellt.

Der Vergleich eines Elements der Kontrollflußmatrix mit der globalen Variable, die die Funktionsnummer enthält, ist wesentlich aufwendiger als die Überprüfung, ob er von Null verschieden ist. Deshalb wurde bei normalen Knoten auf diese Überprüfung verzichtet. Stattdessen wird die volle Überprüfung nur beim Verlassen der Funktion durchgeführt, so daß möglicherweise ein illegaler Sprung in die falsche Funktion erst nach einer verlängerten Verzögerung erkannt wird[1]. Da durch diese Vereinfachung für die Mehrzahl der Knoten ein Zugriff auf die globale Variable f entfällt, ergibt sich ein deutlich geringerer Laufzeitmehraufwand.

[1] Bei den Fehlerinjektionstests (siehe Abschnitt 4.3.2) wurde keine Veränderung der Fehlerüberdeckung durch diese Maßnahme beobachtet.

4.2.3 Beispiel

Der Präprozessor generiert für das Beispielprogramm aus Abbildung 4.3 die Ausgabe in Abbildung 4.4.

```
example()            /* 3rd function */
{
    /* v0 */ static unsigned char __cfm[5][5] = {
        0, 0, 0, 0, 0,
        3, 0, 0, 3, 0,
        0, 3, 0, 0, 0,
        0, 3, 3, 0, 0,
        3, 0, 0, 3, 0,
    };
    unsigned register __sign = 0;
    unsigned char   __fn_sav = __fnum;
    __fnum = 3;
    while (a < 10) {
        /* v1 */ if (!__cfm[1][__sign]) __cfc_err();
        __sign = 1;
        if (b == 0) {
            /* v2 */ if (!__cfm[2][__sign]) __cfc_err();
            __sign = 2;
            c = a;
        }
        /* v3 */ if (!__cfm[3][__sign]) __cfc_err();
        __sign = 3;
        a++;
    }
    /* v4 */ if (__cfm[4][__sign] != __fnum) __cfc_err();
    __fnum = __fn_sav;
}
```

Abbildung 4.4: Ausgabe des Präprozessors

Am ersten Knoten v_0 (SOF) schiebt der Präprozessor ein Codesegment ein, das die Kontrollflußmatrix deklariert und initialisiert. Für jeden Zugriff in die Kontrollflußmatrix wird die jeweils vorhergehende Signatur als Spaltenindex und die aktuelle Signatur als Reihenindex verwendet. Funktionsnummern werden in sequentieller Reihenfolge zugewiesen. Im Beispiel ist die Funktion `example()` die dritte Funktion im Programmtext, also erhält sie die Nummer 3. Die Funktionsnummer 0 darf nicht verwendet werden, da die Null bereits einen unerlaubten Programmfluß anzeigt. Vor dem Aktualisieren der Funktionsnummer f (globale Variable `__fnum`), wird die Nummer der aufrufenden Funktion in der lokalen Variable `__fn_sav` gesichert.

Der Programmcode, der bei den normalen Knoten eingeschoben wird (bei v_1, v_2 und v_3) liest ein Element der Kontrollflußmatrix. Falls der Wert nicht von Null

verschieden ist, wird die Funktion `__cfc_err()` aufgerufen um das Programm zu terminieren.

Am EOF-Knoten v_4 wird, falls der Programmfluß *und* die Funktionsnummer korrekt sind, die Nummer der aufrufenden Funktion, die am Anfang gespeichert wurde, wiederhergestellt.

4.3 Messungen

Zur Bewertung des Verfahrens wurden Messungen auf verschiedenen Rechnersystemen mit verschiedenen Testprogrammen durchgeführt. Um möglichst aussagekräftige Ergebnisse für die Messungen des Laufzeitmehraufwands (und für die Anwendung in Multiprozessoren) zu erhalten, wurden rechenintensive Applikationen für die Tests ausgewählt. Bei den Testprogrammen handelte es sich um leicht veränderte Dhrystone- und Whetstone-Benchmarks (Ganzzahl- bzw. Fließkommaoperationen), um ein Mehrgitterverfahren zur Lösung der zweidimensionalen Poisson-Differentialgleichung (vgl. Abschnitt 2.1), sowie um ein Mehrgitterverfahren zur Lösung der Navier-Stokes-Gleichung. Die Versuche wurden auf i486SX- (Sequent Symmetry), MC68020- (Sun 3/60), Sparc- (Sun 4/65) und MC88100- (MVME188, Einzelrechner des MEMSY-Multiprozessors) Prozessoren durchgeführt. In allen Fällen wurde der gleiche Compiler (`gcc` Version 2.2.2) eingesetzt. Alle zur Verfügung stehenden Codeoptimierungen wurden angewendet. Die Optimierung verringert zwar die gesamte Laufzeit der Programme, aber der relative Laufzeitmehraufwand durch die Selbstüberwachung wird erhöht.

Die verwendeten Testprogramme wurden (bis auf das Whetstone-Programm) auch in den Messungen in [Mic92] unter ähnlichen Bedingungen verwendet. Die Meßergebnisse dieses Abschnitts können also mit den Ergebnissen dieser Arbeit verglichen werden.

4.3.1 Mehraufwand

Durch das Anlegen der Kontrollflußmatrix und durch das Einfügen der Codefragmente zur Selbstüberwachung vergrößert sich der statische Speicherbedarf eines Programms. Beide Einflüsse werden durch den Statement-Faktor und durch die Reduktion beeinflußt.

In den Tests wurden die Faktoren 1, 2, 5 und 10 verwendet (SF-1, SF-2, SF-5 bzw. SF-10). Zusätzlich wurden auch Tests mit aktivierter Reduktion durchgeführt, jeweils mit den Faktoren 2, 5 und 10 (RF-2, RF-5 und RF-10). Für das Dhrystone-Programm ergab sich für die verschiedenen Rechner ein um etwa 50% (RF-10) bis 350% (SF-1) größerer Objektcode, beim Whetstone-Programm lag der Mehrbedarf bei 37–590%, beim Poisson-Mehrgitterverfahren bei 55–270%, sowie im Navier-

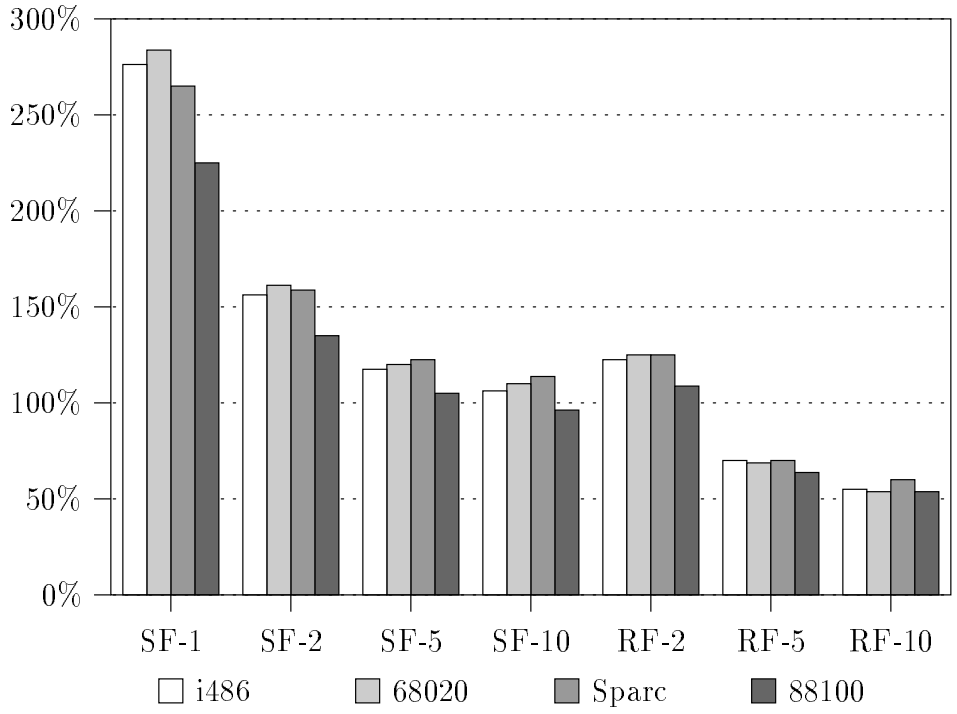

Abbildung 4.5: Speichermehraufwand für verschiedene Rechner (Poisson)

Stokes-Mehrgitterverfahren bei 65–500%. Den Speichermehraufwand bei verschiedenen Parametern für das Poisson-Mehrgitterverfahren zeigt Abbildung 4.5. Der Objektcode des Originalprogramms (100%) belegt etwa 4 kByte[2].

Für diese Grafik wurde nur die Länge des Objektcodes berücksichtigt. Das Poisson-Mehrgitterverfahren allokiert zur Laufzeit (unabhängig vom Statement-Faktor) weitere 3,5 MB Speicherplatz. Im Verhältnis zum insgesamt belegten Speicher beträgt der Mehraufwand für das Poisson-Programm maximal etwa 2% (SF-1).

Der Laufzeitmehraufwand hängt ebenfalls vom Statement-Faktor ab. Die Ergebnisse werden aber erheblich deutlicher durch den Typ des Rechners beeinflußt. Für das 88100-System liegen die Werte im Bereich von 39–83% (Dhrystone), 35–104% (Whetstone), 10–31% (Poisson) und 5–13% (Navier-Stokes). Dabei schneiden die künstlichen Programme (Dhrystone und Whetstone) erheblich schlechter ab. Der Mehraufwand beim Navier-Stokes Verfahren ist deshalb so gering, da die innerste Iterationsschleife sehr kompakt codiert ist. Selbst bei SF-1 entfallen dadurch sehr viele Maschinenbefehle auf je einen Knoten des Programmflußgraphen. Abbildung 4.6 zeigt den Laufzeitmehraufwand des Poisson-Mehrgitterverfahrens für alle Rech-

[2]Die exakten Meßwerte zu dieser Grafik wie auch zu den folgenden Grafiken dieses Abschnitts können den Tabellen in Anhang A.3 entnommen werden.

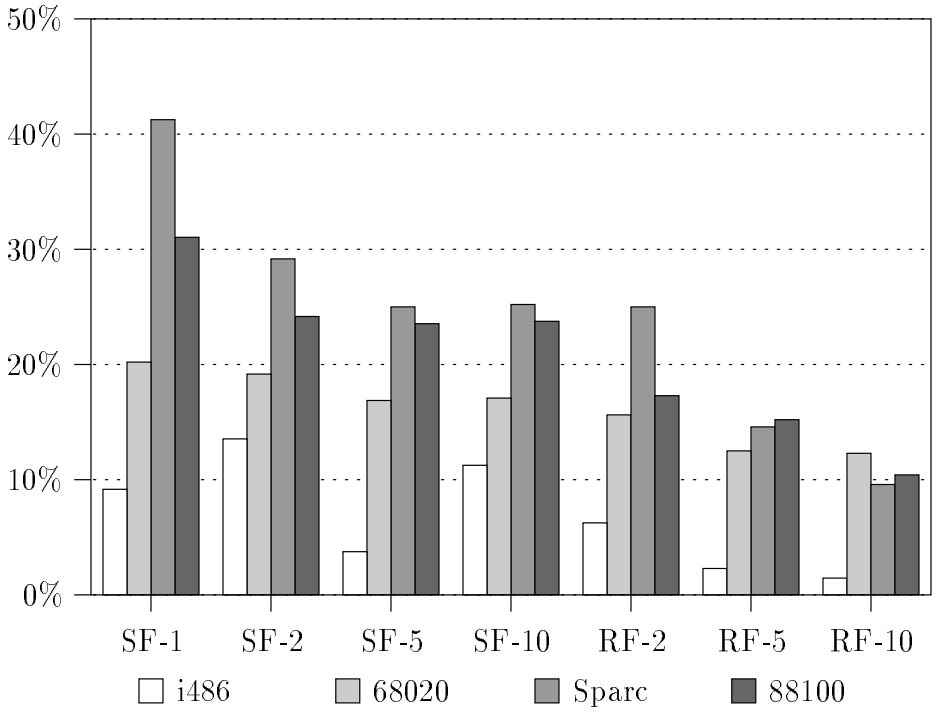

Abbildung 4.6: Laufzeitmehraufwand für verschiedene Rechner (Poisson)

nersysteme im Test. Die absoluten Laufzeiten des Originalprogamms (100%) lagen bei 63 Sekunden (i486), 156 Sekunden (68020), 22 Sekunden (Sparc) und 19.7 Sekunden (88100).

Bei den Laufzeitdaten fällt auf, daß sie einigen zunächst unerklärlichen Schwankungen unterworfen sind. Beispielsweise steigt der Mehraufwand beim i486-System bei Erhöhen des Statement-Faktors von SF-5 auf SF-10 wieder erheblich an. Dieser Effekt ist vermutlich auf das Flattern von Cachezeilen zurückzuführen: Bei SF-10 erreicht die Dimension der Kontrollflußmatrizen zweier häufig verwendeter Unterfunktionen die Dimension 16×16. Das Hinzufügen einer zusätzlichen, mit Null besetzten Zeile und Spalte (die neue Größe der Kontrollflußmatrix ist nicht mehr eine Zweierpotenz) senkt den Laufzeitmehraufwand auf etwa 3%.

Das i486-System zeigt einen deutlich geringeren Laufzeitmehraufwand. Ein möglicher Grund dafür ist die in diesem System fehlende Fließkommaeinheit. Ein relativ hoher Anteil der Laufzeit wird für die Fließkommaemulation verwendet. Für Programme mit hohem Fließkommaanteil ergibt sich daher eine höhere Anzahl von Maschinenbefehlen pro Überwachungsschritt.

Im Vergleich mit den in [Mic92] ermittelten Daten zeigt es sich für das 88100-System, daß das Kontrollflußselbstüberwachungsverfahren einen geringeren Zeit-

mehraufwand bewirkt als eine Implementierung mit Hilfe eines Watchdogprozessors. Der Grund dafür ist die Übertragung der Signaturen über die Speicherschnittstelle zum Watchdogprozessor, die im Vergleich langsamer ist als die Selbstüberprüfung.

4.3.2 Fehlerüberdeckung

Zur Messung von Fehlerüberdeckung und -latenz von CPU-Fehlern wurde ein Fehlerinjektor auf Basis der UNIX `ptrace()`-Schnittstelle verwendet [Sie94]. Diese Schnittstelle, die ursprünglich für die Implementierung von Debuggern vorgesehen ist, ermöglicht Zugriff auf den Adreßraum und die Registerinhalte eines Programms. Fehler können allerdings nicht in den Zustand von externen Geräten oder in das Betriebssystem injiziert werden. Mit dieser Schnittstelle war es möglich, Fehler ohne Modifizierung des Testprogramms in einen laufenden Prozeß zu injizieren.

In den Tests wurden Einbitfehler der Prozessorregister nachgebildet. Zu einem zufälligen, gleichmäßig über den Programmablauf verteilten Zeitpunkt wird dazu ein einzelnes Bit im Zielregister invertiert. Testläufe wurden für Einbitfehler im Programmzähler (Kontrollflußfehler), und für Fehler in einem zufälligen anderen Register (Registerfehler) durchgeführt[3]. Nach dem Invertieren eines Bits in einem Register ist nicht sichergestellt, daß sich diese physikalisch vorhandene Fehlerursache tatsächlich im Sinne der Definition in Abschnitt 2.3 als ein echter Fehler erweist. Es ist möglich, daß das veränderte Register unmittelbar nach der Fehlerinjektion wieder vom Programm überschrieben wird.

Da bereits weit verbreitete Standard-Fehlererkennungstechniken einen großen Anteil aller Fehler erkennen, wurde nicht die Fehlerüberdeckung der Kontrollflußselbstüberwachung gemessen, sondern nur die Überdeckung, die sich bei Superposition mit den Standardverfahren ergibt. Folgende Reaktionen des Systems wurden beobachtet:

- Signal: Eines der Standardverfahren hat einen Fehler erkannt. Der Prozeß hat ein Signal erhalten und wurde dadurch terminiert (vgl. Abschnitt 3.4.1). Typische Beispiele sind Segmentverletzung, Busfehler, illegaler Opcode oder Arithmetikfehler (z.B. Division durch Null).

- Kontrollflußfehler: Ein selbstüberwachendes Programm hat einen Fehler erkannt und die Ausführung abgebrochen.

- Fehler nicht erkannt: Die Programmausführung wurde nach dem Überschreiten einer Zeitschranke terminiert, oder es ergab sich ein anderes Rechenergebnis als im fehlerfreien Fall.

[3]Für die Injektion von Fehlern in den Programmzähler beim MC88100-Prozessor wurden Fehler sowohl in den „fetch program counter“ (fpc) und in den „next program counter“ (npc) injiziert. Danach wurde das Ergebnis gemittelt um einen Vergleich mit den anderen Rechnern zu ermöglichen.

– Kein Fehler: Die injizierte Fehlerursache hatte keine Auswirkung. Das Ergebnis der Berechnung war korrekt und kein Fehler konnte entdeckt werden.

Jede injizierte Fehlerursache wurde nur genau einer der vier Gruppen zugeordnet. Ob ein Fehler, der ein Signal auslöst, zu einem späteren Zeitpunkt eine Kontrollflußabweichung bewirkt hätte, wurde in den Versuchen nicht bestimmt. Abbildung 4.7 zeigt die Häufigkeit der verschiedenen Reaktionen für das Poisson-Mehrgitterverfahren bei Einbitfehlern im Programmzähler. Abbildung 4.8 ist die entsprechende Darstellung für Einbitfehler in einem zufälligen Register.

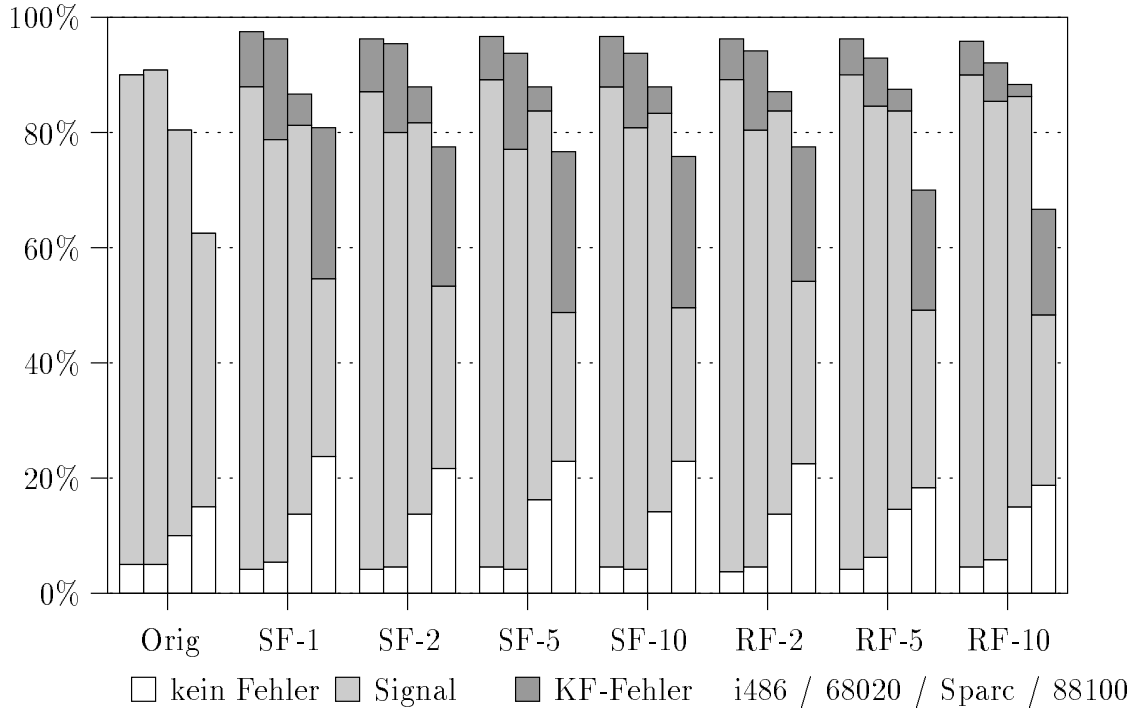

Abbildung 4.7: Systemreaktionen bei Programmflußfehlern (Poisson)

Die Gruppe der mit „Orig“ beschrifteten Meßwerte ist das Ergebnis des Referenzlaufs ohne Überwachung. Die anderen Meßwerte wurden mit Selbstüberwachung gewonnen. In jeder Gruppe von Meßwerten sind (von links nach rechts) die Ergebnisse der Tests auf dem i486-, dem 68020-, dem Sparc- und dem 88100-System aufgetragen.

Die überwältigende Mehrheit der Programmflußfehler wurde durch die Standardtechniken (Signale) erkannt. Einen besonders hohen Anteil hatten dabei die Segmentverletzungen. Da die Fehler in einer zufälligen Bitposition injiziert wurden, ist es nicht überraschend, das etwa 50% der Fehler auf Segmentverletzungen (`SIGSEGV`) entfallen, da nur die Hälfte der Adreßbits bei einem 64kByte großen Programm in einem 2^{32} Byte großen Adreßraum verwendet werden. Das MVME188-System bildet aber eine unrühmliche Ausnahme. Etwa 128 Mbyte des Adreßraums sind für

den Stackbereich reserviert, der bei einem Zugriff auf diese Adressen automatisch vergrößert wird. Um ein Problem in der Signalbehandlung zu umgehen, wird vom Betriebssystem auf dem Stack selbstmodifizierender Code verwendet, d.h. die 128 MByte des Adreßraums sind obendrein als ausführbar gekennzeichnet. Sprünge in diesen Adreßbereich führen also *nicht* zu einer Segmentverletzung, entsprechend ist der Anteil der Signale an der gesamten Fehlerüberdeckung erheblich geringer.

Ein weiterer großer Anteil entfällt auf Busfehler (`SIGBUS`), beispielsweise wegen Wortzugriffen auf eine ungerade Adresse. Nicht alle Betriebssysteme kennen dieses Signal, da es nicht dem POSIX-Standard entspricht. Im Sequent-System (i486) führt diese Art von Fehlern ebenfalls zu einer Segmentverletzung.

Bei etwa einem Zehntel der Fehler wird durch das Erkennen eines illegalen Befehls das `SIGILL`-Signal ausgelöst. Auch das `SIGEMT`-Signal (Emulator Trap), das auf Sun3-Systemen auftritt, ist auf diese Ursache zurückzuführen. Arithmetikfehler stellen einen weiteren geringen Anteil (`SIGFPE`). Die Ergebnisse sind im Anhang A.3 tabelliert.

Bei den beiden RISC-Prozessoren ist die Wahrscheinlichkeit größer, daß sich ein Fehler überhaupt nicht auf das Ergebnis auswirkt (etwa 20% gegenüber 5%). Der Anteil der unerkannten Fehler (der fehlende Abschnitt der Balken bis zur 100%-Linie) wird durch die Selbstüberwachung insbesondere für das 88100-System deutlich reduziert, nämlich von etwa 40% (Orig) auf etwa 20% (SF-1).

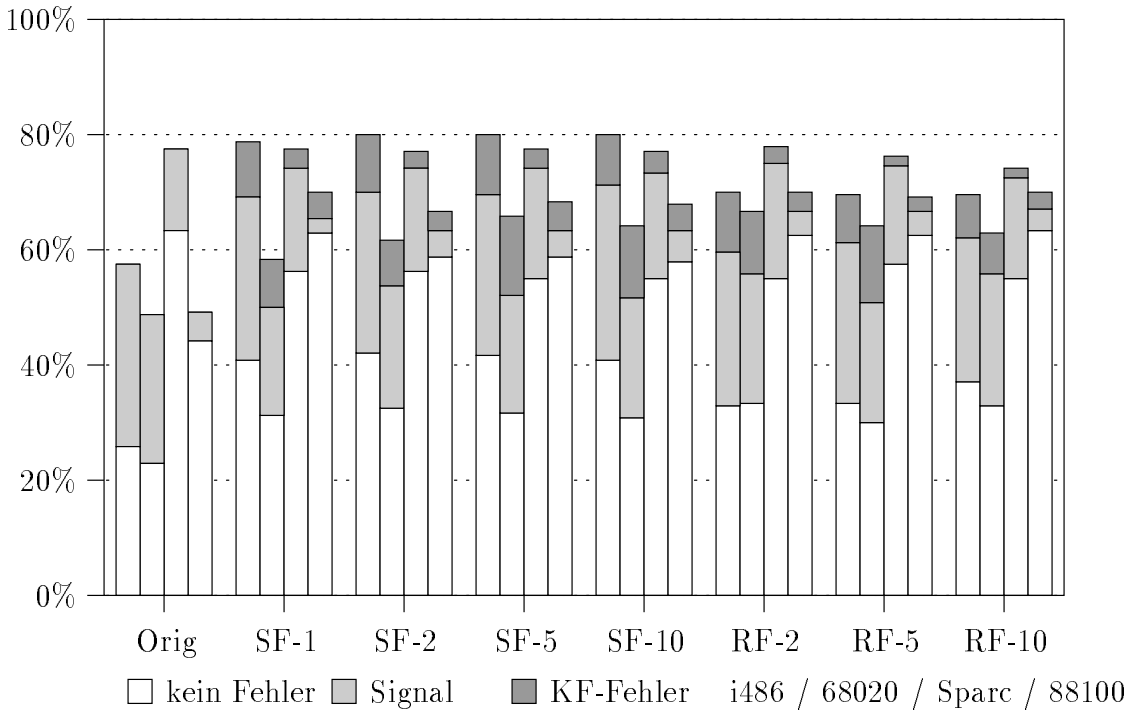

Abbildung 4.8: Systemreaktionen bei Registerfehlern (Poisson)

Im Registertest bleibt für das Sparc- und 88100-System die Mehrzahl der injizierten Fehler ohne Auswirkung. Obwohl die Selbstüberwachung nur wenige dieser Fehler erkennen kann, wird die Wahrscheinlichkeit, daß sich eine Fehlerursache auswirkt, für einige Systeme geringfügig reduziert. Ein möglicher Grund ist, daß die Register auch von der Selbstüberwachung verwendet werden und deshalb öfter neu geladen werden, so daß Fehler häufiger ohne Auswirkung überschrieben werden. Je nach Programmgröße und Datengröße können sich die Ergebnisse bei den Registerfehlern mehr oder weniger stark unterscheiden. Bei einem großen Datenbereich darf auf einen weit größeren Adreßraum zugegriffen werden, ohne daß eine Segmentverletzung ausgelöst wird.

Die Experimente wurden insgesamt je 3000 mal wiederholt. Damit befinden sich alle Ergebnisse mit einer Wahrscheinlichkeit von 90% innerhalb von $\pm 1.5\%$ der tatsächlichen Werte.

4.3.3 Fehlerlatenz

Als ein Teil der Experimente wurde die Fehlerlatenz durch den Fehlerinjektor gemessen. Leider haben die Systemuhren der betrachteten UNIX-Systeme nur eine Auflösung von 1/60 Sekunde, so daß es sehr schwierig war, Aussagen über Zeiten im Bereich von Mikrosekunden zu treffen. Für Sun3-Systeme war es allerdings möglich, im Einzelschritt-Modus die Instruktionszyklen bis zum Erkennen eines Fehlers zu zählen. Aus den vorliegenden Daten kann für das Dhrystone-Programm ein Mittelwert der Fehlerlatenzen der verschiedenen Erkennungsmethoden bestimmt errechnet werden. Auf eine Aufstellung verschiedener Statement-Faktoren wurde aber verzichtet, da kein verwertbarer Unterschied zu erkennen ist:

Methode	Kontrollflußfehler	Registerfehler
ohne Selbstüberwachung	21 Zyklen	73 Zyklen
mit Selbstüberwachung (SF-1)	42 Zyklen	195 Zyklen
davon Signal	39 Zyklen	206 Zyklen
davon KF-Fehler	64 Zyklen	171 Zyklen

Abbildung 4.9 zeigt schließlich die Verzögerung der Fehlererkennung von Einbitfehlern im Programmzähler für das Dhrystone-Programm. Aufgetragen ist der Anteil der unerkannten Fehler über der Zahl der seit der Injektion des Fehlers ausgeführten Befehlszyklen. Ein Datensatz beschreibt die Fehlerlatenzen der Standardmethoden ohne Selbstüberwachung, der andere die Latenzzeiten mit Selbstüberwachung (SF-1).

Die meisten Fehler werden bereits im ersten Befehlszyklus erkannt, weil der Zugriff auf einen ungültigen Datenbereich bzw. auf einen ungültigen Opcode erfolgt. Mit Selbstüberwachung bleibt im Dhrystone-Programm ein zusätzlicher Anteil von etwa

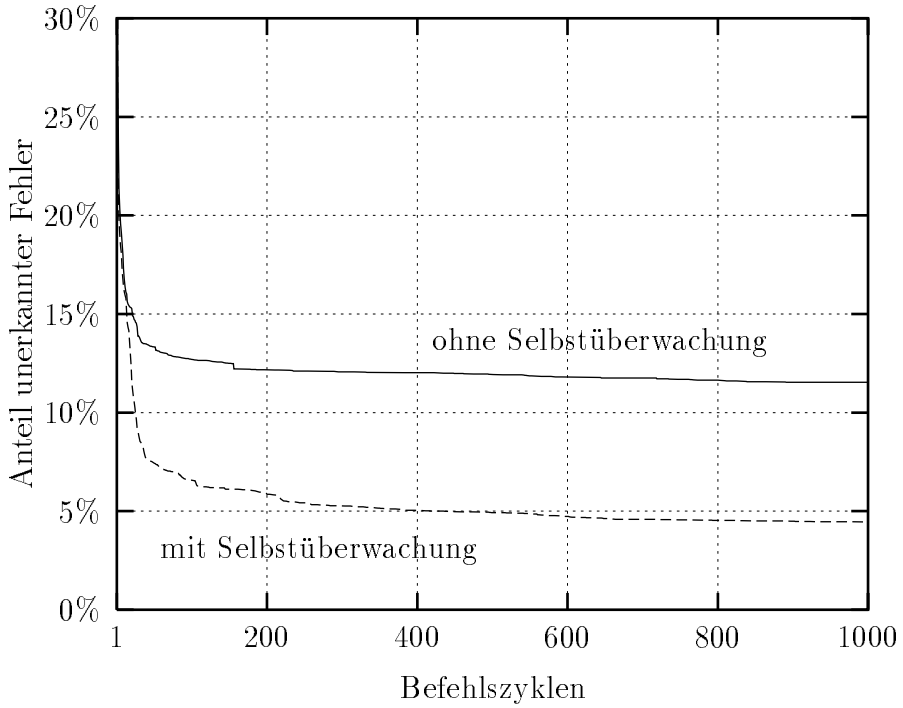

Abbildung 4.9: Verteilung der Fehlerlatenzen

10% der Fehler von vornherein ohne Auswirkung. Aus diesem Grund liegen diese Daten etwa 10% unter den Daten für die Standardverfahren. Nach etwa 1000 Befehlszyklen wird praktisch kein Fehler mehr erkannt und ein Restanteil unerkannter Fehler bleibt zurück.

4.4 Bewertung

Um etwas über die Verbesserung der Fehlerüberdeckung durch das Verfahren aussagen zu können, müssen die Meßergebnisse des vorherigen Abschnitts in irgendeiner Form bewertet werden. Das Nachbilden von physikalischen CPU-Fehlern durch Einbitfehler in einem Prozessorregister ist aber eine nur unzureichende Modellvorstellung. Hardwarefehler treten bevorzugt während der Umschaltvorgänge auf, z.B. durch eine Störung beim Zustandswechsel eines Bits im Programmzähler von null auf eins. Um das dynamische Verhalten also zu berücksichtigen, sollten bei der Bewertung von Einbitfehlern im Programmzähler diejenigen Bits, die im tatsächlichen Betrieb ihren Zustand häufig wechseln, stärker gewichtet werden. Diese Gewichtung ist vom Lokalitätsverhalten der Programme abhängig. Für Registerfehler ist eine Gewichtung der einzelnen Bits nicht möglich. Häufig werden Fehler auch nicht den Inhalt der Register betreffen, sondern beispielsweise die Auswahl eines anderen

Registers bewirken. Zu Fehlerinjektionstests mit einer komplexeren Modellierung [TA80, BA84] sind weitere Arbeiten im Gange [SPS+94].

Dennoch soll versucht werden, aus den Messungen des letzten Abschnitts wenigstens eine Aussage über die grobe Tendenz zu gewinnen. Zur Berechnung einer Gesamtüberdeckung wurde die Fehlerüberdeckung der Programmzählertests (Kontrollflußfehler) gegenüber den Ergebnissen für die Registerfehler im Verhältnis zwei zu eins gewichtet, da sich die Mehrheit der CPU-Fehler, wie eingangs erwähnt, als Abweichung des Programmflusses äußert. Das Ergebnis dieser Berechnung wurde wiederum für alle Testprogramme gemittelt. Anschließend wurde die Verbesserung der durchschnittlichen Fehlerüberdeckung gegen den durchschnittlichen Laufzeitmehraufwand aufgerechnet, denn durch die längere Laufzeit sind Fehler während des Programmlaufs wahrscheinlicher. Das Ergebnis dieser Berechnung kann durch den *Gütefaktor* F des Verfahrens ausgedrückt werden.

Zur Motivation des Gütefaktors F sei zunächst ein Rechner betrachtet, in dem Fehler mit der Rate λ auftreten, die Fehlerüberdeckung gleich c ist, und in dem ein Rechenauftrag nach Ablauf der Zeit T berechnet ist. Unter Einsatz eines zusätzlichen Fehlererkennungsverfahrens möge sich eine neue (größere) Fehlerüberdeckung c' und eine neue (längere) Programmlaufzeit T' ergeben. Möglicherweise kann sich auch die Anfälligkeit des Systems gegenüber Fehlern verändern, aber für Softwareverfahren sei angenommen, daß die Fehlerrate λ gleich bleibt. Das Verfahren führt allenfalls dann zu einer Verbesserung der Fehlertoleranzeigenschaften, wenn wenigstens die Vertrauenswürdigkeit eines Rechenergebnisses durch seinen Einsatz erhöht wird, d.h. wenn gilt (vgl. Formel 3.2):

$$V' = e^{-(1-c')\lambda T'} \quad > \quad e^{-(1-c)\lambda T} = V$$

Durch Umformen dieser Ungleichung ergibt sich:

$$\frac{1-c}{1-c'} \cdot \frac{T}{T'} \quad > \quad 1$$

wobei die linke Seite der Ungleichung, der Bruch

$$F = \frac{1-c}{1-c'} \cdot \frac{T}{T'}$$

im folgenden als der Gütefaktor des Verfahrens bezeichnet wird. Da das Verhalten von $e^{-(1-c)\lambda T}$ für $(1-c)\lambda T \approx 0$ sehr gut durch eine lineare Funktion angenähert werden kann, entspricht der Gütefaktor F der proportionalen Veränderung des maximal zulässigen Rechenzeitbedarfs einer Applikation, bevor eine vorgegebene Vertrauenswürdigkeit $V(t) > 90\%$ unterschritten wird (vgl. Abbildung 3.2, Seite 30). Sinkt beispielsweise nach einer Berechnung der Länge 2 MTTF die Vertrauenswürdigkeit

eines Systems unter 90%, so kann durch Einsatz eines Fehlererkennungsverfahrens mit dem Gütefaktor $F = 2$ eine Berechnung mit dem Rechenzeitbedarf 4 MTTF mit derselben Vertrauenswürdigkeit ausgeführt werden. Der Rechenzeitbedarf von 4 MTTF ist dabei auf das Originalsystem bezogen, je nach Zeitmehraufwand des Fehlererkennungsverfahrens ist die tatsächliche Rechendauer auf dem System mit zusätzlicher Fehlererkennung größer.

Dieser Gütefaktor wurde für die verschiedenen Rechnersysteme und Parameter für das Selbstüberwachungsverfahren ermittelt. Das Ergebnis dieser Rechnung ergibt die Gütefaktoren F_{SF-1} bis F_{RF-10}, die in Abbildung 4.10 aufgetragen sind. Werte größer als eins steigern die Vertrauenswürdigkeit einer Berechnung, Werte kleiner als eins verringern sie.

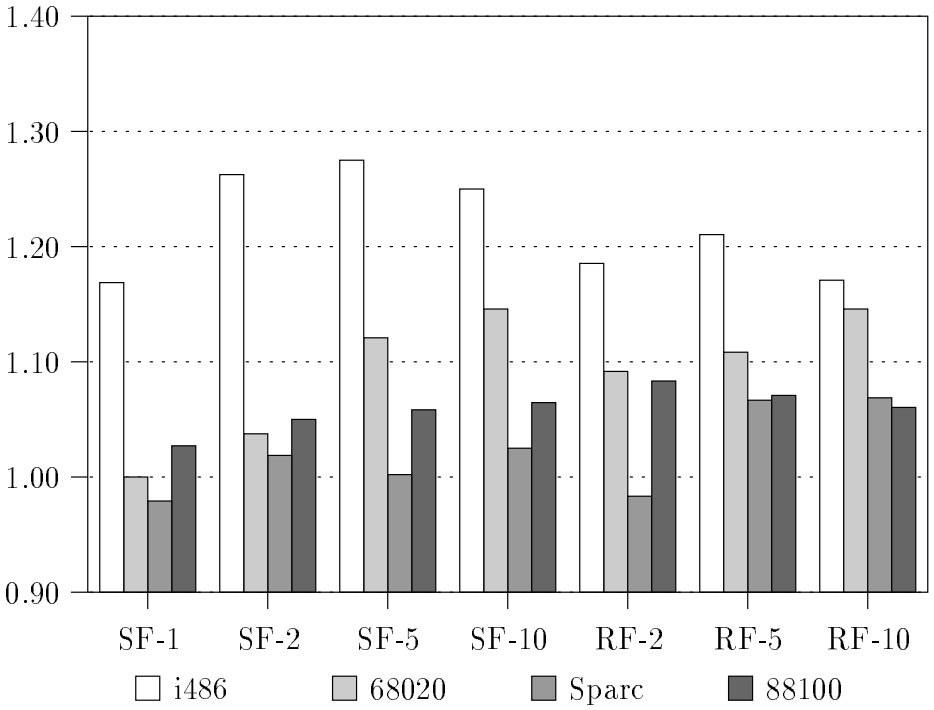

Abbildung 4.10: Gütefaktor der Selbstüberwachung

Insgesamt ergibt sich ein eher uneinheitliches Bild. In einigen Fällen ist der Gütefaktor kleiner als eins, d.h. die Selbstüberwachung war eher schädlich als nützlich. Für diesen Effekt sind die hohen Laufzeitverluste durch die Selbstüberwachung der beiden künstlichen Programme (Dhrystone und Whetstone) verantwortlich. Für die Mehrzahl der Fälle kann aber davon ausgegangen werden, daß die Anzahl der unerkannten Fehler durch den Einsatz von Kontrollflußselbstüberwachung reduziert wird. Allgemein scheinen größere Werte für den Statementfaktor und das Aktivieren der Reduktion einen höheren Gütefaktor zu ermöglichen, der allerdings nur im Bereich

um 1,1 liegt. Bei einem Statement-Faktor von z.B. RF-5 ist je nach Rechnersystem ein Laufzeitmehraufwand im Bereich von 13–30% zu erwarten.

Die Kontrollflußselbstüberwachung sowie die Kontrollflußüberwachung durch einen Watchdogprozessor nach dem SIC-, ESIC- oder SEIS-Verfahren allein (in Kombination mit den Systemmethoden) erscheinen aber nicht ausreichend, um in massiv parallelen Rechnern eine ausreichende Erhöhung der Fehlerüberdeckung zu bewirken. Selbst eine geringe Verbesserung der Fehlerüberdeckung muß mit sehr hohem zusätzlichen Aufwand erkauft werden. Die mittlere Fehlerlatenz dieser Verfahren liegt aber bei Bruchteilen einer Sekunde und ist damit weit unter dem geforderten Eckwert von einer Sekunde.

Kapitel 5

Zustandssicherung und Fehlerbehebung

Für die globale Zustandssicherung ist der programmgesteuerte Ansatz weitgehend portabel und gegenüber den transparenten Verfahren schneller. Es hat sich als sinnvoll erwiesen (vgl. Abschnitt 3.3.3), mit Hilfe von Koordinationsprotokollen, z.B. der Zweiphasen-Freigabe, die Konsistenz der Sicherungspunkte gegenüber Fehlern zu schützen.

Durch die Synchronisierung bei der globalen Zustandssicherung verringert sich die Nebenläufigkeit paralleler Applikationen. Im hier beschriebenen Ansatz wird versucht, diesen Nachteil auszugleichen. Durch nebenläufige Zustandssicherung wird globale Synchronisierung vermieden, außerdem können Wartezeiten beim Verwenden eines langsamen Sicherungsspeichers maskiert werden. Zur nebenläufigen Speicherung werden bei der Implementierung im MEMSY-Rechner Funktionen der virtuellen Speicherverwaltung des Betriebssystems verwendet.

Die Kenntnis der Arbeitsweise der programmgesteuerten Zustandssicherung ist für das Verständnis der Koordinationsprotokolle und des Sicherungsverfahrens notwendig. Erweiterungen dieser Verfahren, die für die transparente Zustandssicherung notwendig sind, werden in Abschnitt 5.5 nur kurz erwähnt. Eine exaktere Beschreibung der Protokolle, die Einbindung der Fehlererkennung, sowie die Beschreibung ihrer Umsetzung im MEMSY-Rechner, folgt erst im nächsten Kapitel.

5.1 Programmgesteuerte Zustandssicherung

Einen besonders kritischen Punkt der Umsetzung von Rückwärtsfehlerbehebungsverfahren bildet der Zugriff auf externe Objekte (vgl. Abschnitt 2.4.1). Für die programmgesteuerte Rückwärtsfehlerbehebung bedeutet dies, daß vom Programmierer gewisse Regeln befolgt werden müssen.

5.1.1 Regeln für die Programmierung

Als wichtigste Regel sollten nach dem Verändern eines externen Objekts (z.B. einer Ausgabedatei) keine Sicherungspunkte mehr erstellt werden. Das Rücksetzen auf einen solchen Sicherungspunkt kann sonst zum Verlust der vor dem Sichern ausgegebenen Daten führen. Wird nach der ersten Ausgabe kein Sicherungspunkt mehr erstellt, so kann im Fehlerfall auf den letzten Sicherungspunkt zurückgesetzt werden, um die Ausgabedatei nochmals zu öffnen und vollständig neu zu erzeugen. Damit die Rücksetzverluste gering bleiben, sollte die Ausgabe erst am Ende des Programms erfolgen.

Weiter ist zu beachten, daß eine Applikation unmittelbar nach dem Start noch keine Fehler tolerieren kann. Erst nach der Initialisierung und nach dem Erstellen des ersten Sicherungspunkts kann zurückgesetzt werden. Bis zum Schreiben des ersten Sicherungspunkts sollte nur geringer Rechenfortschritt erzielt werden, so daß ein Neustart der Applikation nur mit einem geringen Verlust verbunden ist.

Den Programmablauf einer fehlertoleranten, numerischen Applikation kann man also wie folgt in drei Teile gliedern:

- In der **Anlaufphase** besteht noch kein Schutz gegenüber Fehlern. Die Fehlertoleranzfunktionen müssen initialisiert und parametrisiert werden. Diese Phase sollte möglichst kurz sein, da die Fehlerbehebung einen Neustart des Programms bedeutet.

- Die **Arbeitsphase** beginnt sobald der erste Sicherungspunkt erstellt ist. Die Applikation läuft im fehlertoleranten Betrieb und kann auf den jeweils letzten Sicherungspunkt zurückgesetzt werden.

- Erst mit Beginn der **Ausgabephase** der Ergebnisse dürfen externe Objekte manipuliert werden. Da keine Sicherungspunkte mehr erstellt werden, sollte diese Phase ebenfalls möglichst kurz sein.

Nicht immer können Arbeitsphase und Ausgabephase voneinander getrennt werden. Wenn beispielsweise das Verhalten eines physikalischen Systems für eine Folge von diskreten Zeitpunkten berechnet werden soll, müssen möglicherweise Zwischenergebnisse nach jeder Iteration bereits ausgegeben werden. Für sequentielle Schreibzugriffe, die in numerischen Anwendungen fast ausschließlich auftreten, könnte in jedem Sicherungspunkt der Schreibzeiger der Datei mitgesichert werden, beim Wiederaufsetzen können von dieser Stelle ab die fehlerhaften Daten wieder überschrieben werden. Sicherer ist es, jeweils verschiedene Ausgabedateien zu erzeugen.

5.1.2 Aufgaben einer Bibliothek

Das Erstellen fehlertoleranter Programme auf Parallelrechnern wird durch häufig wiederkehrende Vorgehensweisen charakterisiert, die durch Bibliotheksfunktionen unterstützt werden können. Funktionsbereiche für die sich eine solche Unterstützung anbietet, sind das physikalische Übertragen der Daten von und zum Sicherungsspeicher, sowie die Koordinierungsprotokolle, die die Konsistenz der Sicherungspunkte beim Sichern und beim Wiederanlauf sicherstellen:

- Das Format der Sicherungspunkte muß festgelegt werden, speziell welche Daten im Sicherungspunkt enthalten sind.
- Es sind Programmstellen zu identifizieren, an denen keine Kommunikation zwischen den Prozessen stattfindet. Ggf. muß synchronisiert werden.
- Programmcode zum konsistenten Sichern ist zu erstellen. In diesen Bereich fallen die physikalische Ansteuerung des Sicherungsspeichers und die Koordinationsprotokolle, um Fehler beim Sichern zu tolerieren.
- Es muß Programmcode bereitgestellt werden, der im Fehlerfall die Fortsetzung der Berechnung mit den Sicherungsdaten ermöglicht. Dazu gehört das physikalische Lesen der Sicherungsdaten und das Synchronisieren der Applikation zur Rücksetztrennnung.

Das Format eines Sicherungspunktes muß es ermöglichen, bestimmte Daten zu kennzeichnen, die nach dem Wiederanlauf für die Berechnung des Ergebnisses notwendig sind. Dazu kann im Programm eine Objektliste erstellt werden, die alle Daten beschreibt, die gesichert werden müssen. Die Beschreibung eines Objekts kennzeichnet den Speicherbereich, der durch dieses Objekt belegt wird und kann beispielsweise aus der Adresse des Objekts und seiner Größe bestehen.

Die Objektliste muß beim Wiederanlauf zum Zurücklesen der Daten neu aufgebaut oder eingelesen werden werden. Die erstere der beiden Methoden ist sicherer, da sich die Adressen der Objekte beim Wiederholungslauf geändert haben können: Dynamisch durch `malloc()` angeforderte Speicherbereiche können beispielsweise auf anderen Adressen liegen. Entsprechend kann auch beim Abspeichern und Wiederherstellen von Zeigervariablen beim Wiederanlauf eine Aufbereitung nötig sein.

Für den Wiederanlauf sind zwei verschiedene Strategien vorstellbar. Zum einen kann man vereinbaren, daß das Programm im Falle eines Fehlers zunächst abgebrochen wird. Zur Fehlerbehebung wird das Programm neu aufgerufen. Das Programm stellt fest, daß Sicherungspunkte geladen werden können, und verhält sich entsprechend, d.h. Initialisierungsschritte, die nur bei einem Neustart von Anfang an ausgeführt werden müssen, können übersprungen werden, die Sicherungsdaten werden geladen,

und schließlich kann anhand der Sicherungsdaten festgestellt werden, an welcher Stelle das Programm fortzusetzen ist.

In der anderen Strategie wird das Programm nicht abgebrochen, sondern es wird vom Programmierer eine Fehlerbehandlungsroutine spezifiziert. Im Fehlerfall wird die Programmausführung dieser Routine übergeben, die dann wiederum den Sicherungspunkt laden und die Berechnung fortsetzen kann. Dieser Ansatz ist unsicherer und funktioniert nicht, wenn durch den Fehler der Stack, die Speicherabbildung oder Programmcode zerstört wurden. Soll dieser Ansatz überhaupt funktionieren, ist er nicht mehr portabel, denn es müssen, wie bei der transparenten Zustandssicherung, Änderungen im Betriebssystem vorgenommen werden.

Bei der Zustandssicherung kann das Verhalten der Applikationen berücksichtigt werden. Numerische Applikationen sind meistens als eine mehr oder weniger große Iterationsschleife aufgebaut. Zunächst werden lokal neue Werte berechnet, die im Anschluß daran mit den anderen Prozessen ganz oder teilweise ausgetauscht werden. Nachdem ein Prozeß den Datenaustausch abgeschlossen ist, herrscht für die Dauer der nächsten Iteration Ruhe in seiner Kommunikation, also ist die Gelegenheit günstig, einen neuen Sicherungspunkt zu erstellen. Der Programmierer muß darauf achten, daß alle Prozesse gleichzeitig im Sinne der „$\rightarrow$“-Relation (siehe Abschnitt 2.4.2.1) neue Sicherungspunkte erstellen, damit ein konsistenter Sicherungspunkt entsteht.

Für das eigentliche Schreiben und Lesen der Sicherungspunkte muß der Sicherungsspeicher angesprochen werden. Koordinierungsprotokolle kommen zum Einsatz, um die Konsistenz bei der Zustandssicherung und die Rücksetztrennung sicherzustellen (vgl. Abschnitte 5.2 und 5.4). Parameter der Funktionen zum Schreiben und Lesen ist die Objektliste, die das Format der Sicherungsdaten beschreibt. Techniken zum Unterstützen von nebenläufiger Zustandssicherung (Abschnitt 5.3) können portabel und ohne Eingriffe in das Betriebssystem ebenfalls auf Bibliotheksebene realisiert werden.

Wenn im Verlauf der Berechnung neue sicherungswürdige Daten entstehen, kann es sinnvoll sein, eine Veränderung des Sicherungsformats während der Arbeitsphase zuzulassen. Zu beachten ist aber, daß der Speicherplatz für die Daten eines Sicherungspunktes beim Einlesen der Daten bereits allokiert sein muß. Verändert sich die Größe (bzw. das Format) eines Sicherungspunkts im Verlauf der Rechnung, muß vor dem Einlesen bekannt sein, welche Daten im Sicherungspunkt enthalten sind und wieviel Speicher dafür benötigt wird. Damit das Programm vor dem Einlesen ggf. den notwendigen Speicher reservieren kann, ist in der MEMSY-Fehlertoleranzbibliothek[1] ein Sicherungspunkt durch eine vom Programmierer bestimmte Formatkennung gekennzeichnet. Vor dem Einlesen kann das Programm die Formatkennung abfragen. Anschließend kann im Programm der notwendige Speicher allokiert und die Objektliste erzeugt werden.

[1] Das Online-Manual der für MEMSY realisierten Bibliothek steht im Anhang A.2.

5.1.3 Anwendungsbeispiel

Als Beispiel für die programmgesteuerte Vorgehensweise zeigt Abbildung 5.1 die Umgestaltung eines Iterationsverfahrens zu einem fehlertoleranten Programm.

```
main ()
{
    /* Initialisieren */
    /* Festlegen des Sicherungsformats */
    if (Wiederanlauf) {
        /* Einlesen der Sicherungsdaten */
    } else {
        /* Einlesen der Anfangswerte */
    }
    do {
        /* Iteration */
        /* Datenaustausch */
        /* Sicherungspunkt speichern */
    } while (noch nicht fertig);
    /* Ergebnisse ausgeben */
    /* Terminierung */
}
```

Abbildung 5.1: Beispiel eines fehlertoleranten Programms

Beim Programmstart überprüft das Programm durch den Aufruf einer Bibliotheksfunktion, ob es sich um einen Wiederholungsversuch handelt. Falls nein, werden nach der Initialisierung die Anfangswerte eingelesen. Da die Sicherungspunkte im Verlauf der Rechnung in diesem Beispiel immer das gleiche Format haben, kann bereits von Anfang an das Format der Sicherungspunkte festgelegt werden, wie oben beschrieben in Form einer Liste von Adressen und Längenangaben. Dieses Format wird sowohl beim Lesen als auch beim Schreiben von Sicherungspunkten verwendet. Damit ist die Anlaufphase abgeschlossen.

Die Iterationsschleife entspricht der Arbeitsphase. Nach der Iteration, dem Senden der eigenen Randwerte und dem Empfangen der Randwerte der Nachbarn ist für die Dauer der nächsten Iteration zunächst keine Kommunikation zu erwarten. Also kann an dieser Stelle ein Sicherungspunkt geschrieben werden.

Am Ende der letzten Iteration beginnt die Ausgabe der Ergebnisse. In der Ausgabephase werden keine Sicherungspunkte mehr erstellt. Zum Abschluß des Programms können die Sicherungspunkte gelöscht werden.

Nach einem Fehler wird das Programm zum Wiederanlauf einfach neu gestartet. Wenn das Programm die erste Iteration bereits abgeschlossen hatte, so wird es

Abbildung 5.2: Dezentrales Zweiphasen-Freigabeprotokoll

Die Teilnehmer treffen ihre Entscheidung über den vorläufigen Sicherungspunkt unabhängig voneinander, je nach den Informationen, die sie lokal erhalten haben. Dabei gelten die folgenden Entscheidungsregeln:

> *Sobald von allen anderen Teilnehmern* `READY`*-Nachrichten empfangen wurden, führe eine Commit-Operation aus.*
>
> *Falls einer der Teilnehmer einen Fehler gemeldet hat, führe eine Abort-Operation aus.*

Das Protokoll ist nicht „independently recoverable", d.h. es kann der Fall auftreten, daß ein Prozeß sich beim Wiederanlauf in einem unentschiedenen Zustand befindet. In seinem Sicherungsspeicher liegt ein vorläufiger Sicherungspunkt, für den er vor dem Ausfall noch kein Commit oder Abort ausführen konnte. Er muß andere Teilnehmer nach der richtigen Entscheidung fragen.

Die Entscheidung, ob ein dezentrales Protokoll einem zentralen Ansatz vorzuziehen ist, kann von den Eigenschaften des Kommunikationssystems abhängig gemacht werden. Ist ein Rundspruch von etwa vergleichbarem Aufwand wie eine einzelne Nachricht (z.B. in lokalen Netzwerken belegt eine Nachricht und ein Broadcast die gleiche Bandbreite), so liegt der Vorteil auf der Seite des dezentralen Ansatzes, denn die zentrale Zweiphasen-Freigabe benötigt insgesamt bei n Teilnehmern $4n$ Nachrichten, der verteilte Ansatz aber nur $2n$ Rundsprüche. Auch bei anderen Verbindungstopologien kann sich ein dezentrales Protokoll als günstig erweisen, beispielsweise in einem Torus-Netzwerk wie im MEMSY-Multiprozessor (siehe Abschnitt 6.2.2).

5.2.2 Anforderungen an den Rundspruch

Es ist eine Voraussetzung des obigen Verfahrens, daß alle Rundsprüche zum einen zuverlässig, zum anderen in der gleichen Reihenfolge zu jedem Teilnehmer übertragen werden. Die erste Bedingung ist notwendig, denn würde ein Teilnehmer eine `READY`-Nachricht verpassen, so würde er entweder ewig warten oder nach einer Zeitschrankenüberschreitung mit einer Abort-Operation terminieren.

Nachrichtenverluste können entweder toleriert oder durch geeignete Übertragungsverfahren maskiert werden. In Hochleistungs-Parallelrechnern sind Maskierungsverfahren aber wegen des hohen Mehraufwands, der dabei in der Kommunikation entsteht, nicht sinnvoll. Deshalb wird eine zweigleisige Strategie verfolgt: Systemnachrichten zur Koordinierung werden zuverlässig übertragen, während die normale Kommunikation auf Anwendungsebene durchaus von Fehlern betroffen sein darf. Auf Anwendungsebene auftretende Kommunikationsfehler werden als Prozessorfehler eines der Endpunkte betrachtet und führen zu Fehlermeldungen oder Ausnahmen (vgl. Abschnitt 3.4.1).

Abbildung 5.3: Fehler durch unterschiedliche Nachrichtenreihenfolge

In dieser Abbildung erstellen drei kooperierende Prozesse ihre vorläufigen Sicherungspunkte und senden durch einen Rundspruch `READY`. Anschließend fahren sie mit dem Bearbeiten der Applikation fort. Prozeß P_2 ist kurze Zeit später fehlerhaft und erzeugt eine Fehlernachricht. Da sonst kein Fehler auftritt, werden die Prozesse P_1 und P_2, mit der letzten `READY`-Nachricht die Entscheidung Commit treffen. Für Prozeß P_3 ergibt sich ein kritischer Fall, da die Fehlernachricht und die `READY`-Nachricht in der Reihenfolge durch zufällige Kommunikationsverzögerungen vertauscht sind. Beim Eintreffen der Fehlermeldung müßte er nach der obigen Entscheidungsregel eigentlich das Sicherungsprotokoll mit Abort beenden.

Einen ähnlichen Fall zeigt Abbildung 5.3b. P_3 führt als einziger ein Commit aus, da er bereits alle `READY`-Nachrichten erhalten hat, noch bevor er die Fehlermeldung erhält.

Zur Abhilfe gibt es verschiedene Möglichkeiten. Zum einen kann man die Reaktion auf Fehler verzögern, damit die `READY`-Nachricht noch eintrifft, bevor P_3 auf den Fehler reagiert. In dieser Methode muß eine Zeitschranke für die maximale Übertragungsdauer angegeben werden. Die Überwachung von Zeitschranken sollte aber vermieden werden, denn die minimale Länge einer Zeitschranke ist schwer zu bestimmen, darüberhinaus sind Systeme, die Zeitschranken einsetzen, nur schwer zu skalieren und zu warten. Ohne Zeitschranke an dieser Stelle ist darüberhinaus eine schnellere Fehlerbehandlung möglich.

Eine andere Möglichkeit wäre es, für eine global einheitliche Nachrichtenreihenfolge zu sorgen. Global einheitliche Nachrichtenreihenfolge ist eine nur sehr aufwendig zu realisierende Eigenschaft. Ein Beispiel für eine Software-Umsetzung findet sich in [CASD85]. Der Algorithmus verwendet ebenfalls Zeitschrankenüberwachung und erfordert lose synchronisierte Uhren.

Eine weitere, dritte Möglichkeit ist die lokale Reihenfolgetreue, in der nur Nachrichten der selben Quelle überall in der selben Reihenfolge ankommen müssen. Diese Eigenschaft ist erheblich einfacher umzusetzen, z.B. mittels Numerierung der Nachrichten. Stellt der Empfänger eine Lücke in der Reihe der empfangenen Nachrichten fest, muß er auf die fehlende Nachricht warten, bevor er spätere Nachrichten verarbeiten darf.

Um ein korrektes Arbeiten des Protokolls sicherzustellen, muß aber die im letzten Abschnitt aufgestellte Bedingung für die Entscheidung „Abort“ umformuliert werden:

> *Falls einer der Teilnehmer einen Fehler meldet, ohne vorher* `READY` *gesendet zu haben, führe eine Abort-Operation aus.*

Im Beispiel Abbildung 5.3a wird P_3, wenn die Fehlernachricht eintrifft, anhand der Nachrichtennummer erkennen, daß noch eine Nachricht dieses Senders fehlt und erst einmal abwarten. Auf das Beispiel Abbildung 5.3b angewendet, können nun Prozesse P_1 und P_2 nicht für Abort entscheiden, da eine `READY`-Nachricht der `ERROR`-Nachricht vorausging, vielmehr werden beide, wenn von P_3 die `READY`-Nachricht endlich eingetroffen ist, ein Commit ausführen.

Schließlich ist es eine vierte Möglichkeit, auf eine Fehlermeldung sofort mit dem Abbruch des Programms zu reagieren und das Koordinationsverfahren erst beim Wiederanlauf, wenn alle Teilnehmer wieder einsatzbereit sind, zu wiederholen.

In den Protokollen (Abschnitt 6.1.1) wird ein Verfahren eingesetzt, das Elemente der dritten und vierten Methode enthält. Das Warten auf Nachrichten mit kleinerer Nummer wird vermieden. Auch das Problem, daß sich in diesem Verfahren Teilnehmer nach dem Wiederanlauf in einem unentschiedenen Zustand befinden können, wird auf eine sehr einfache Art gelöst.

5.3 Nebenläufige Zustandssicherung

5.3.1 Nebenläufige Zweiphasen-Freigabe

Wenn das Zweiphasen-Freigabeprotokoll mit einer Commit-Operation abschließt, so ist eine *globale* Synchronisation der gesamten Anwendung vorausgegangen. Alle Teilnehmer haben zu diesem Zeitpunkt ihren neuen Sicherungspunkt gespeichert, falls einer der beteiligten Prozesse langsamer als die anderen ist, so muß auf diesen Prozeß gewartet werden.

Durch Auswahl eines geeigneten Zeitpunkts zum Sichern kann eine Stelle gefunden werden, in der die Applikation in ihrer Kommunikationsstruktur eine Synchronisation zum Datenaustausch durchführt (Abschnitt 5.1). Oft ist aber nur eine *lokale*

Synchronisation notwendig, wenn die Daten nur mit dem direkten Nachbarn ausgetauscht werden. Ein Beispiel zeigt Abbildung 5.4.

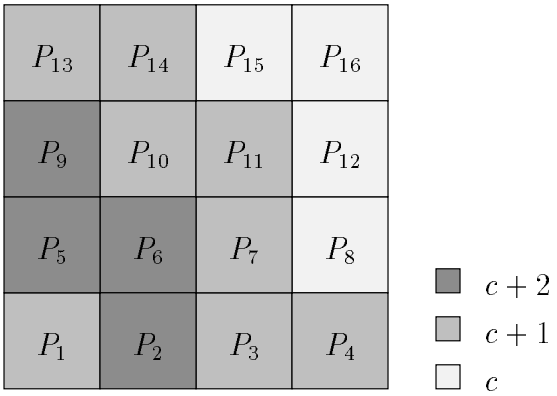

Abbildung 5.4: Unterschiedlicher Rechenfortschritt in einer Applikation

Die Prozesse $P_{2,5,6,9}$ sind am weitesten fortgeschritten und haben die c+2-te Iteration abgeschlossen. Mit der nächsten Iteration können sie aber noch nicht beginnen, da die Daten der $c+2$-ten Iteration von den Nachbarn noch nicht vorliegen. Ebenso muß beispielsweise P_{11} noch auf die Ergebnisse der Iteration $c+1$ seiner Nachbarn P_{12} und P_{15} warten, bevor er mit der Iteration $c+2$ beginnen kann. Im Beispiel hat sich auf diese Weise ein Abstand von 2 Iterationen ausgebildet.

Falls die Laufzeiten jeweils einer Iteration stochastischen Schwankungen unterworfen sind, gibt es in den Applikationen trotz lokaler Synchronisation noch eine gewisse Schwankungsbreite. Durch die Zweiphasen-Freigabe würde diese Schwankungsbreite in einer fehlertoleranten Version nicht mehr vorhanden sein. Da der Fortschritt der Rechnung nicht vom Ergebnis der Zustandssicherung abhängt, kann aber die Zweiphasen-Freigabe auch nebenläufig durchgeführt werden.

Sobald der Sicherungspunkt lokal erstellt ist, wird eine `READY`-Nachricht verschickt und das Programm unmittelbar danach fortgesetzt. Nachrichten des Koordinierungsprotokolls werden asynchron empfangen und asynchron ausgewertet. Sobald die Bedingung erfüllt ist, kann ebenfalls asynchron die Commit-Operation ausgeführt werden. Bei einer Abort-Operation wird das Hauptprogramm unterbrochen und die Fehlerbehebung eingeleitet.

Bei nebenläufiger Zweiphasen-Freigabe kann der Fall auftreten, daß bereits eine Schreibanforderung für den nächsten Sicherungspunkt vorliegt, obwohl das Verfahren für den vorherigen Sicherungspunkt noch nicht abgeschlossen ist. Das Verhalten in diesem Fall hängt davon ab, wieviele Sicherungspunkte gleichzeitig im Sicherungsspeicher aufbewahrt werden können. Durch das Verwenden eines Zweiphasen-Freigabeprotokolls hat die Zustandssicherung bereits den zweifachen Bedarf an stabilem Speicher als ohne: für einen vorläufigen Sicherungspunkt und für den aktuellen

Abbildung 5.5: Belegung zweier Puffer für Sicherungspunkte

5.3.2 Nebenläufiges Sichern

Es ist naheliegend, außer der Zweiphasen-Freigabe, auch das eigentliche Speichern asynchron durchzuführen, so daß währenddessen die Applikation fortgesetzt wird. Dadurch kann der Einfluß von Wartezeiten beim Zugriff auf langsame Speichermedien, wie beispielsweise auf einen zentralen Server, verringert werden. Der einfachste Ansatz ist das Puffern aller Sicherungsdaten im Hauptspeicher. Allerdings wird bei hoher Auslastung der Resourcen durch die Applikation nicht ausreichend viel Speicher zum Puffern eines kompletten Sicherungspunkts zur Verfügung stehen. Weiterhin darf die `READY`-Nachricht für die Zweiphasen-Freigabe aus Gründen der Zuverlässigkeit erst dann erzeugt werden, wenn die Sicherungsdaten vollständig den Sicherungsspeicher erreicht haben. Die Zeit für das physikalische Sichern t_{ss} ist entsprechend erheblich höher als der Zeitmehraufwand für das Sichern t_s (vgl. Abschnitt 3.3.3).

Neben der Pufferung sind auch verschiedene Verfahren vorgestellt worden, die die virtuelle Speicherverwaltung verwenden [LNP90, BP91, BP92]. Die Sicherungsdaten werden zunächst nur schreibgeschützt. Ein Schreibzugriff des Arbeitsprozesses ist also zunächst blockiert. Beispielsweise im UNIX-Betriebssystem (oder

in Mach-basierten Betriebssystemen) kann ein solcher Datenbereich das „copy-on-write"-Attribut erhalten [Bac86]. Ein solcher Speicherbereich ist nur zum Lesen freigegeben, bei einem Schreibzugriff wird ein Seitenfehler („page fault") ausgelöst. Bei der Behandlung des Seitenfehlers erstellt das Betriebssystem eine Kopie der entsprechenden Seite. Die Speicherabbildung des Prozesses, der den Seitenfehler ausgelöst hat, wird verändert, die neue Kopie der Speicherseite wird Bestandteil seiner privaten und beschreibbaren Daten.

Mit der Originalkopie eines Speicherbereichs kann verschieden umgegangen werden. Die Speicherseiten verbleiben entweder im virtuellen Speicher des Rechnersystems und werden je nach Bedarf auf den Hintergrundspeicher ausgelagert („virtueller Sicherungspunkt"), oder ein anderer Prozeß kann im Hintergrund die Daten auf den Sicherungsspeicher übertragen und den belegten Speicher anschließend freigeben.

Das Verwenden der virtuellen Speicherverwaltung hat den Vorteil, daß nicht alle Sicherungsdaten auf einmal in einen Puffer umkopiert werden müssen. Stattdessen werden die Daten nur nach Bedarf dupliziert. Bei Programmen mit großer Lokalität der Schreibzugriffe wird dieser Ansatz der Pufferung überlegen sein, andernfalls wird sich wegen der vielen Seitenfehler eine relative Verschlechterung gegenüber der Pufferung einstellen. Der Sicherungsprozeß kann zu einer Entspannung der Speichersituation beitragen, indem er den Schreibschutz für jede Speicherseite jeweils nach dem Sichern sofort wieder aufhebt. Solche Optimierungen erfordern aber den Eingriff in das Betriebssystem und sind nicht portabel.

Das „copy-on-write"-Attribut ist ebenfalls nicht unter der unmittelbaren Kontrolle des Programmierers. Als einfache Implementierung konnte für die MEMSY-Fehlertoleranzbibliothek auf den `fork()`-Aufruf zurückgegriffen werden. Dieser Aufruf erzeugt einen neuen Sohn-Prozeß, der mit seinem Vater den Speicher gemeinsam nützt. Dabei erhält aber der gesamte Speicher das „copy-on-write"-Attribut, es ist nicht möglich, nur die Sicherungsdaten auszuwählen.

Die Vorbedingung zur Auswahl eines Sicherungszeitpunkts bei programmgesteuerter Zustandssicherung war Ruhe in der Kommunikation zu diesem Zeitpunkt. Sind auch gemeinsame Speicherbereiche im Sicherungspunkt enthalten, bedeutet dies, daß während dem Sichern kein anderer Prozeß auf den gemeinsamen Speicher schreibend zugreifen darf.

5.4 Fehlerbehebung

Alle Teilnehmer müssen beim Wiederanlauf, um Inkonsistenzen zu vermeiden (Abschnitt 2.4.2.2), vor dem Start synchronisiert werden. Zur Synchronisation kann, wie bei der Zweiphasen-Freigabe, ein dezentraler Ansatz gewählt werden. Statt auf `READY`-Nachrichten wird auf `SYNC`-Nachrichten aller Teilnehmer gewartet. Wenn ein Teilnehmer nach einem Fehler bereit ist zum Wiederanlauf, so sendet er eine

SYNC-Nachricht. Wie bei der Zweiphasen-Freigabe ist auch für die Synchronisation die Nachrichtenreihenfolge kritisch. Für eine Synchronisationsnachricht muß sicher feststehen, ob sie sich auf einen Zustand *vor* oder *nach* einem Fehler bezieht.

Wenn die Prozesse selbst die Träger von Nachrichten sind, also das Kommunikationssystem selbst keine Nachrichten enthalten kann (das Kommunikationssystem ist zustandslos), so sind nach der Synchronisation zum Wiederanlauf alle alten Nachrichten vernichtet. Diese Annahme trifft oft nicht zu, denn Nachrichten können für eine gewisse Zeit in Puffern und Warteschlangen verzögert werden. Als eine einfache Abhilfe kann man wiederum Zeitschranken verwenden: Vor dem Wiederanlauf läßt man eine gewisse Zeit verstreichen, in der alle Nachrichten, die ankommen, vernichtet werden. Dabei können beträchtliche Wartezeiten notwendig sein. In [Ech90] wird dieses Vorgehen „Anhaltetrennung“ genannt.

Bei der „Kennzeichentrennung“ wird an alle Nachrichten ein eindeutiges Kennzeichen angehängt, das den Rücksetzversuch identifiziert. Dies kann durch *global konsistentes* Aufwärtszählen der Rücksetzkennung jeweils beim Auftreten eines Fehlers bewerkstelligt werden. Das Verfahren ist in Abschnitt 6.1.1 genauer beschrieben.

Bei der nebenläufigen Zustandssicherung kann zum Zeitpunkt eines Fehlers im System der Rechenfortschritt verschieden sein. Vor dem Rücksetzen nach einem Fehler in einem fortgeschrittenen Prozeß kann noch auf Nachzügler gewartet werden: Wenn im Beispiel (Abbildung 5.5) der P_1 ausfällt, so hat er immerhin den Sicherungspunkt $c+1$ erfolgreich gespeichert, dieser Rechenfortschritt muß nicht durch Rücksetzen auf den Sicherungspunkt c vernichtet werden. Das Rücksetzen wird verzögert, bis auch P_3 den Sicherungspunkt $c+1$ geschrieben hat.

Eine andere Version des Wiederanlaufverfahrens, das beim Wiederanlauf Konfigurationsänderungen zuläßt, ist in [Hön94] beschrieben.

5.5 Transparente Zustandssicherung

Bei der transparenten Zustandssicherung ist die Unterbrechung aller Prozesse zu beliebigen Zeitpunkten möglich. Es entsteht also nicht, wie bei der programmgesteuerten Zustandssicherung, ein Verlust durch das Synchronisieren auf einen bestimmten Stand des Rechenfortschritts. Allerdings muß auf Ruhe in der Kommunikation gewartet werden.

Auch die transparente Zustandssicherung kann nebenläufig durchgeführt werden, um Verzögerungen durch langsame Sicherungsspeicher oder durch das Warten auf Koordinierungsnachrichten zu vermeiden. Wie bei der transparenten Zustandssicherung kann der „copy-on-write“ Mechanismus verwendet werden. Das relativ einfache Zustandssicherungsprotokoll aus Abschnitt 5.2.1 muß aber noch mit Zwischenstufen ergänzt werden. Abbildung 5.6 zeigt einen Ablaufplan. Eine ausführlichere Beschreibung befindet sich in [Sie93].

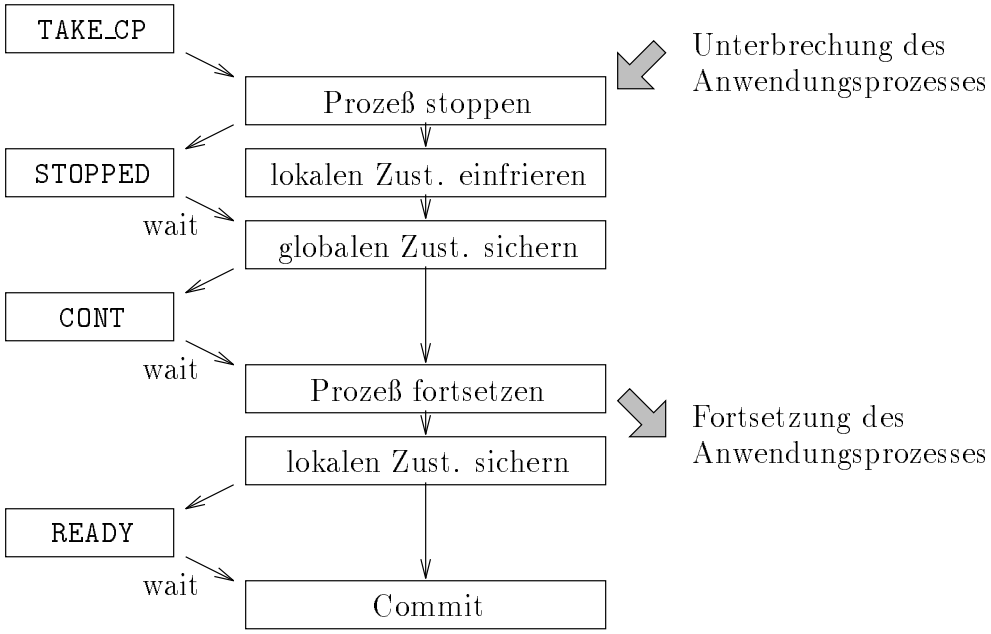

Abbildung 5.6: Ablauf der transparenten, nebenläufigen Zustandssicherung

Für speichergekoppelte Systeme stellen gemeinsame Speicher, auf die von verschiedenen Rechnerknoten zugegriffen werden kann, ein Problem dar. Im MEMSY-System beispielsweise ist die Speicherverwaltung der einzelnen Knoten autonom. Wenn die Zugriffsrechte in einem Knoten auf „read only“ gesetzt werden, so hat ein anderer Knoten möglicherweise immer noch Schreibberechtigung. Ohne entsprechende Konsistenzprotokolle für die Speicherabbildung aller zugriffsberechtigten Knoten können gemeinsame Speicherbereiche erst nach einer Synchronisation mit den benachbarten Knoten gesichert werden.

Bevor ein Prozeß Daten abspeichern darf, die auch anderen Prozessen zugänglich sind („globaler Zustand“, z.B. Shared-Memory oder Nachrichtenpuffer) muß erst aus Konsistenzgründen sichergestellt sein, daß auch alle anderen Prozesse die Ausführung angehalten haben, und daß keine Nachrichten mehr im System unterwegs sind. Erhält ein Prozeß eine `TAKE_CP`-Nachricht, sei es nach Ablauf eines Zeitintervalls oder durch einen anderen Prozeß, so wird die Ausführung der Applikation gestoppt und zur Bestätigung eine `STOPPED`-Nachricht abgeschickt. Die `STOPPED`-Nachrichten werden für eine Zwischensynchronisierung verwendet. In der Tat ist dieser Schritt komplizierter als in Abbildung 5.6 dargestellt, da, selbst wenn ein Prozeß gestoppt ist, nicht davon ausgegangen werden kann, daß keine Nachrichten mehr unterwegs sind. Man hat hier wiederum die Möglichkeit, eine gewisse Wartezeit verstreichen zu lassen. Eine bessere Möglichkeit ist das Zählen von Nachrichten:

Jeder Prozeß der Applikation besitzt einen Nachrichtenzähler, der auf Null initia-

lisiert wird. Beim Senden einer Nachricht wird der Zähler um eins erhöht, beim Empfangen um eins erniedrigt. Die Summe aller Zähler der Prozesse einer Applikation entspricht dann der Anzahl der abgeschickten aber noch nicht empfangenen Nachrichten. Sobald sich eine Summe gleich Null ergibt, ist keine Nachricht mehr unterwegs. Der Vorgang, der im Bild als das Warten auf `STOPPED`-Nachrichten dargestellt ist, entspricht tatsächlich dem Bilden der globalen Summe der Nachrichtenzähler. Dieser Vorgang muß solange wiederholt werden, bis sich die Summe Null ergibt.

Vor diesem Schritt kann ein Prozeß bereits seinen lokalen Zustand einfrieren, d.h. mit Schreibschutz und „copy-on-write"-Attribut versehen. Wenn dann die Summe der Zähler Null ergibt, d.h. wenn alle Prozesse angehalten haben und keine Nachrichten mehr unterwegs sind, kann der globale Zustand gesichert werden. Die Applikation darf erst fortgesetzt werden, wenn der globale Zustand aller Prozesse gesichert ist, und zwar nach einer weiteren Zwischensynchronisierung mittels der `CONT`-Nachrichten. Der globale Zustand kann also nicht nebenläufig gesichert werden, sondern muß direkt auf das Sicherungsmedium übertragen oder gepuffert werden.

Parallel zur Anwendung wird im Anschluß an diesen Schritt der lokale Zustand in den Sicherungsspeicher übertragen. Sobald dies fehlerfrei gelungen ist, kann ein Teilnehmer `READY` senden, und der Sicherungspunkt darf, sobald die Bedingung zur Freigabe erfüllt ist, durch eine Commit-Operation als gültig erklärt werden.

Von den zwei zusätzlichen Zwischenschritten ist das Warten auf `CONT`-Nachrichten nur bei nebenläufiger Zustandssicherung wichtig. Das vorherige Warten auf die `STOPPED`-Nachrichten ist auch im Fall von nicht nebenläufiger Zustandssicherung notwendig, da aus Gründen der Konsistenz auf jeden Fall solange gewartet werden muß, bis die Kommunikation ruht.

5.6 Messung der Sicherungszeiten

Beide Zustandssicherungsverfahren wurden im MEMSY-Multiprozessor erprobt und die Sicherungszeiten gemessen. Als Testprogramm wurden zwei Multigrid-Verfahren verwendet. Das Verfahren zur Lösung der Poisson-Gleichung belegte einen Adreßraum von etwa 3,5 MB (Speicherbedarf eines Sicherungspunkts bei transparenter Zustandssicherung). Bei programmgesteuerter Sicherung war die Größe eines Sicherungspunkts bei etwa 1,9 MB (54%). Für das Programm zur Lösung der Navier-Stokes-Gleichung konnte der Speicherbedarf des Sicherungspunkts bei programmgesteuerter Zustandssicherung auf etwa 15% reduziert werden.

Die Speicherzeit für Sicherungspunkte wird durch die Eigenschaften des Sicherungsspeichers und durch die Systemlast beeinflußt. Als Sicherungsspeicher wurde die lokale Festplatte verwendet. Als Dateisystem stand nur das System V Dateisystem

zur Verfügung, das nur schlechte Übertragungszeiten zuläßt, die obendrein sehr stark von der Fragmentierung abhängen[2].

UNIX-Systeme besitzen grundsätzlichen einen „Buffer Cache“, in dem alle Diskzugriffe gepuffert werden. Auch Schreibzugriffe werden grundsätzlich gepuffert, sofern dies nicht beim Öffnen einer Datei ausdrücklich verboten wird. Bei gepuffertem Schreiben kehrt der Schreibaufruf zurück, sobald er alle Daten im Puffer abgelegt hat, in System V (also auch in MEMSOS) besteht danach keine Möglichkeit festzustellen, wann die Daten tatsächlich auf der Festplatte angekommen sind. Bei gepuffertem Schreiben ist dem System eine von den verschiedensten Faktoren (Größe des Buffer-Cache, Fragmentierung des Dateisystems, Systemlast) abhängige Zeit einzuräumen, während der die Daten mit hoher Wahrscheinlichkeit die Festplatte erreicht haben, ehe angenommen werden darf, daß der Sicherungspunkt erfolgreich gespeichert wurde. Aus diesem Grund sollte nur das synchrone Schreiben unter Umgehung des Puffers verwendet werden[3].

Während der Messungen war das Testprogramm das einzige lauffähige Programm in einem MEMSY-Knoten mit 2 Prozessoren. Der freie Arbeitsspeicher (vor Start des Testprogramms) betrug 7 MByte. Die Größe des Buffer-Cache war auf 2 MByte konfiguriert.

Für die Messungen wurde jeweils das selbe, annähernd gleich fragmentierte Dateisystem verwendet. Das Testprogramm war das Poisson-Multigridverfahren. Von den insgesamt acht Kombinationsmöglichkeiten der verschiedenen Techniken (gepuffert oder ungepuffert, nebenläufig oder nicht, sowie transparent oder programmgesteuert) sind fünf Meßergebnisse in Abbildung 5.7 dargestellt.

Bei nebenläufiger, ungepufferter Zustandssicherung können jeweils zwei Werte gemessen werden: Der Mehraufwand, der durch die Zustandssicherung für das Programm entsteht (entspricht t_s in der Terminologie aus Kapitel 3), und der Zeitaufwand, der im Hintergrund tatsächlich benötigt wird, bis der Sicherungspunkt auf dem Sicherungsspeicher abgelegt ist (t_{ss}). Der Zeitaufwand t_{ss} entspricht dabei in etwa dem Mehraufwand t_s beim nicht nebenläufigen Schreiben der Sicherungspunkte. Diese beiden Messungen (nicht nebenläufiges, ungepuffertes Schreiben, programmgesteuert und transparent) wurden deshalb weggelassen.

Die Ergebnisse dieser beiden Messungen sind in den zwei linken, hell unterlegten Balkenpaaren dargestellt: für transparente (T) und programmgesteuerte (PS) Zustandssicherung. Der linke Balken, für den die linke Sekundenskala gilt, zeigt den Mehraufwand, der durch die Zustandssicherung für das Programm entsteht (t_s). Der rechte Balken zeigt den Zeitaufwand, der benötigt wird, bis der Sicherungspunkt

[2] Das System V-Dateisystem der derzeitigen MEMSOS-Version ist nicht mehr Stand der Technik. Modernere Dateisysteme erlauben Transferraten, die um ein Vielfaches höher sind.

[3] In BSD-UNIX-Systemen darf das gepufferte Schreiben verwendet werden: Mit dem `fsync()`-Systemaufruf kann sichergestellt werden, daß alle Daten sicher die Festplatte erreicht haben.

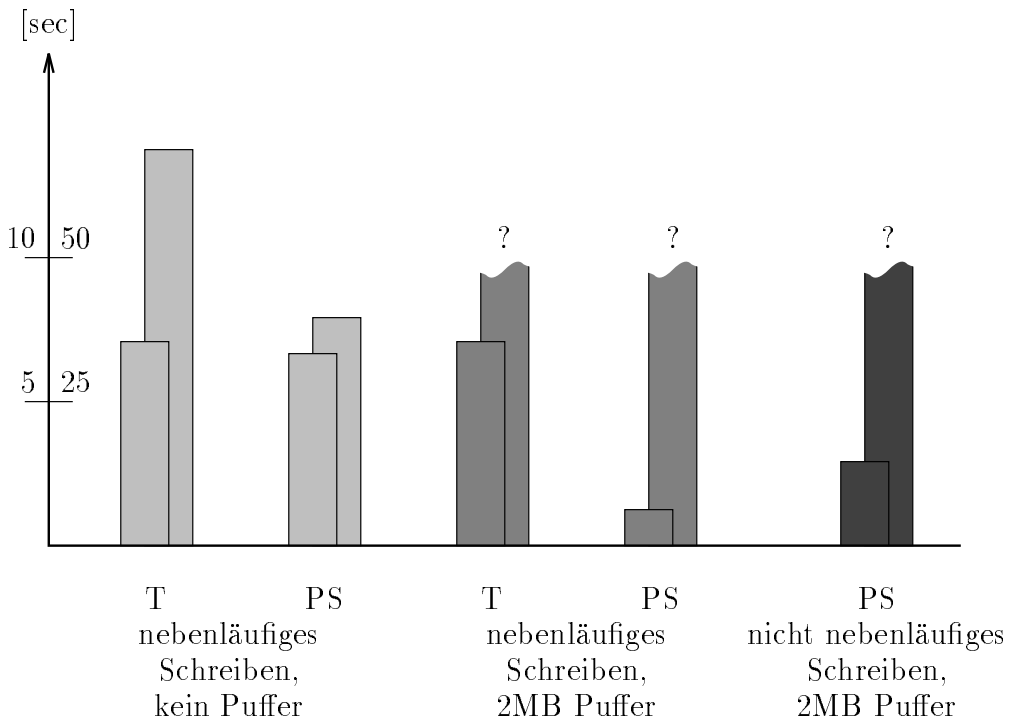

Abbildung 5.7: Transparente und programmgesteuerte Zustandssicherung

tatsächlich auf dem Sicherungsspeicher abgelegt ist (t_{ss}). Für die jeweils rechten Balken gilt die Sekundenskala rechts neben der Zeitachse.

Bei transparenter, nebenläufiger und ungepufferter Zustandssicherung beispielsweise beträgt der Zeitmehraufwand für einen Sicherungspunkt etwa 7 Sekunden, während das Übertragen der Daten auf die Festplatte im Hintergrund in etwa 70 Sekunden in Anspruch nimmt.

Aber auch die programmgesteuerte, nebenläufige und ungepufferte Zustandssicherung bringt zunächst nur wenig Vorteil, obwohl der Sicherungspunkt nur halb so groß ist (zweite Stelle von links, etwa 6,5 Sekunden Mehraufwand). Für das Schreiben im Hintergrund können 40 Sekunden veranschlagt werden. Der geringe Vorteil beim Mehraufwand entsteht dadurch, daß die programmgesteuerte Sicherung, wie in Abschnitt 5.3.2 beschrieben, mittels des `fork()`-Aufrufs implementiert wurde, so daß während der Sicherungsdauer die selben Daten wie im Fall der transparenten Zustandssicherung schreibgeschützt sind und gegebenenfalls dupliziert werden müssen[4].

Bei transparentem, nebenläufigem und gepuffertem Schreiben (dritte Stelle von

[4]Der Datentransfer eines Prozesses auf das Dateisystem ist darüberhinaus in der gegenwärtigen MEMSOS-Version sehr ineffizient

links), sinkt der Mehraufwand nur unerheblich, da der Puffer überläuft. Wie oben erwähnt, kann keine Zeit t_{ss} angegeben werden, innerhalb der die Daten zur Festplatte übertragen sind. Dieser Umstand ist durch das „?“ über dem rechten Balken gekennzeichnet.

Bei nebenläufigem und gepuffertem Schreiben ist die programmgesteuerte Zustandssicherung im Vorteil (vierte Stelle von links), da der kleinere Sicherungspunkt nahezu komplett gepuffert werden kann. Es kann wiederum t_{ss} nicht angegeben werden.

Schließlich ist ganz rechts das Meßergebnis für das programmgesteuerte, nicht nebenläufige, aber gepufferte Sichern dargestellt. Gegenüber dieser Vorgehensweise erlaubt der nebenläufige Ansatz immer noch eine Halbierung von t_s.

In allen Versuchen war sichergestellt, daß im Hauptspeicher noch ausreichend Reserve vorhanden war, damit die Kopien der Speicherseiten im Bedarfsfall ohne Auslagerung auf den Hintergrundspeicher angelegt werden konnten. Wird der Hauptspeicher aber von der Applikation vollständig ausgenutzt, ist die nebenläufige Sicherung für das Testprogramm nicht mehr sinnvoll.

Aufgrund der Vielzahl der nur schwer kontrollierbaren Einflüsse kann Abbildung 5.7 allenfalls als Beweis für die Implementierbarkeit gelten, immerhin lassen die Daten erkennen, daß auch in der Praxis die physikalische Sicherungszeit t_{ss} in etwa linear von der Größe des Sicherungspunkts abhängt (ca. 20 Sekunden pro MByte). Durch nebenläufiges Sichern kann für das Beispielprogramm unter günstigen Bedingungen, bei entsprechender Implementierung auch im Fall von programmgesteuerter Zustandssicherung, die Sicherungszeit t_s immerhin um den Faktor 10 auf etwa 2 Sekunden pro MByte verkleinert werden.

Bei der Wahl des Sicherungsintervalls t_i ist zu beachten, daß die Zeit t_{ss} in ihrer Größenordnung bereits t_i erreichen kann. In solchen Fällen, in denen das optimale Sicherungsintervall kleiner als t_{ss} ist, muß ein schnelleres Speichermedium, z.B. RAM-Speicher, als Sicherungsspeicher eingesetzt werden.

Bei Angaben über die Wiederanlaufzeit müssen mehrere Fälle unterschieden werden. Bei einem Knotenneustart ist für einen MEMSY-Rechnerknoten mit einer Lade- und Startzeit des Betriebssystems von mehreren Minuten zu rechnen. Die genaue Zeit hängt unter anderem auch von der Anzahl und Speicherkapazität der angeschlossenen Plattenlaufwerke ab. Kann ein Fehler durch Neustart des Programms von den Rücksetzpunkten behoben werden, so ist mit einer Ladezeit der Daten (wie beim ungepufferten Schreiben) von etwa 20 Sekunden pro Megabyte zu rechnen. Liegt ein Teil der Daten noch im Buffer-Cache, können sich auch schnellere Zeiten ergeben.

Der Mehraufwand, der durch die Koordinierungsprotokolle entsteht, wird in Kapitel 6 betrachtet.

Kapitel 6

Protokolle zur Rückwärtsfehlerbehebung

Die Funktionsbereiche Fehlererkennung, Zustandssicherung und Fehlerbehebung sind durch geeignete Protokolle zu einem Rückwärtsfehlerbehebungsverfahren zu integrieren. Ein wesentliches Kriterium beim Entwurf ist die Eignung eines solchen Verfahrens für den praktischen Einsatz in einem massiv parallelen Rechnersystem. Insbesondere muß für die Voraussetzungen geprüft werden, inwieweit sie effizient umgesetzt werden können. Viele der häufig geforderten Eigenschaften, z.B. die Erhaltung der Reihenfolge von Nachrichten, sind in realen Systemen beim Vorhandensein von Fehlern nur unter hohem Zeitmehraufwand realisierbar. Oft bedeuten diese Lösungen gerade bei der Kommunikation einen Mehraufwand, der sich für viele Anwendungen besonders störend auf den maximal erreichbaren Speedup auswirkt und auf massiv parallelen Rechnern besonders teuer bezahlt wird.

In nahezu allen Verfahren müssen an irgendeiner Stelle Zeitschranken angenommen werden, die nicht nur dem System angepaßt, sondern auch bei Veränderungen und Erweiterungen modifiziert werden müssen. Die Skalierbarkeit und Wartbarkeit, sowie auch die Portabilität einer Lösung wird durch das Verwenden von Zeitschranken verringert. Als ein Maß für die Güte eines Verfahrens kann betrachtet werden, inwieweit Zeitschranken das Verhalten beeinflussen.

Als weitere Frage stellt sich, inwieweit Dienste zentralisiert oder verteilt angeboten werden. Zentralisierung verringert die Autonomie der Einzelrechner und ist deshalb nicht wünschenswert im Sinne der Fehlertoleranz. Probleme entstehen insbesondere durch die Frage, inwieweit ein zentraler Dienst von allen Prozessoren übernommen werden kann, und wie in Fehlerfällen der Prozessor zu bestimmen ist, der diesen Dienst übernimmt. Der verteilte Ansatz ist im allgemeinen einfacher und übersichtlicher, da alle Prozesse gleich behandelt werden können und weniger Ausnahmen und Sonderbehandlungen berücksichtigt werden müssen. In vielen Fällen ist dieser Ansatz auch weniger aufwendig. Das hier vorgestellte Verfahren verwendet einen verteilten Ansatz, kann aber auch auf ein zentrales Modell übertragen werden.

Aus Gründen der Einfachheit werden nur die Protokolle für die programmgesteuerte Zustandssicherung beschrieben. Die zusätzlichen Probleme bei transparenten Verfahren wurden bereits im vorigen Kapitel angedeutet.

Außerdem sei angenommen, daß nur jeweils ein vorläufiger Sicherungspunkt existieren darf, d.h. es sind nur zwei Sicherungspuffer vorhanden (vgl. Beispiel in Abbildung 5.5). Die Protokolle können leicht erweitert werden, um einen größeren Abstand der einzelnen Prozesse, d.h. mehrere Sicherungspunkte, zu unterstützen.

6.1 Arbeitsweise des Verfahrens

In Abschnitt 5.2.2 hat es sich bereits gezeigt, daß die Nachrichten zur Koordinierung wenigstens der Eigenschaft der lokalen Reihenfolgetreue genügen müssen. Im Verfahren wird allerdings nicht die lokale Reihenfolgetreue angewendet, sondern ein im Prinzip ähnlicher Ansatz. Die Nachrichten bestehen nur aus einer aktuellen Zustandsbeschreibung des Absenders, die dem Empfänger erlaubt, so weit wie nötig eventuelle Zwischenschritte (fehlende Nachrichten) zu rekonstruieren.

Die Reaktion auf Ereignisse, wie das Empfangen von Nachrichten und das Auftreten von Fehlern, ist abhängig vom Kontext, in dem die Ereignisse auftreten. Dieser Kontext ist durch den Zustand eines Teilnehmers definiert.

6.1.1 Zustände eines Prozesses

Ein Teilnehmer (Applikationsprozeß) kann sich in genau einer der drei Phasen „Leerphase“, „Freigabe“ oder „Synchronisation“ befinden:

- In der *Freigabephase* „F“ ist ein vorläufiger Sicherungspunkt vorhanden und ein dezentrales Zweiphasen-Freigabeprotokoll wird abgearbeitet. Die Phase beginnt, sobald ein vorläufiger Sicherungspunkt gespeichert wurde und endet mit einer Commit- oder Abort-Operation.
- Die *Synchronisationsphase* „S“ dient zur Synchronisierung beim Wiederanlauf und wird nach dem Erkennen eines Fehlers (bzw. nach einer Abort-Operation) betreten.
- Der Teilnehmer befindet sich in der *Leerphase* „L“, wenn er sich in keiner der beiden anderen Phasen befindet. Diese Phase wird durch die Commit-Operation oder nach Beheben eines Fehlers (Synchronisation) betreten.

Der Zustand eines Teilnehmers ist nicht nur durch die Phase $p \in \{L, F, S\}$ bestimmt, in der er sich befindet. Weitere Zustandsinformation wird durch zwei Zahlenwerte angegeben:

- Die Sicherungspunktnummer c ist die Nummer des zuletzt erfolgreich geschriebenen vorläufigen Sicherungspunkts;
- die Synchronisationsnummer s beschreibt den aktuellen Wiederholungslauf und wird als Rücksetzkennzeichen verwendet.

Damit ist der Zustand Z eines Prozesses ein Tripel

$$Z = (p, c, s), \quad \text{wobei} \quad p \in \{L, F, S\} \quad \text{und} \quad c, s \in N_0$$

Der Anfangszustand aller Prozesse sei $Z = (L, 0, 0)$. Die Zustandsinformation $Z = (p, c, s)$ eines Teilnehmers werde nicht durch Fehler verfälscht oder zerstört.

Während eines Sicherungszyklus durchläuft ein Teilnehmer die Leerphase und die Freigabephase. Die Freigabephase ist aber nicht mit der Zustandssicherung gleichzusetzen, sondern sie beginnt erst, wenn der vorläufige Sicherungspunkt gespeichert ist. Abbildung 6.1 zeigt als Beispiel einen möglichen Verlauf der Zustandsübergänge eines Prozesses P bei der Freigabe eines Sicherungspunkts und bei der Behebung eines Fehlers:

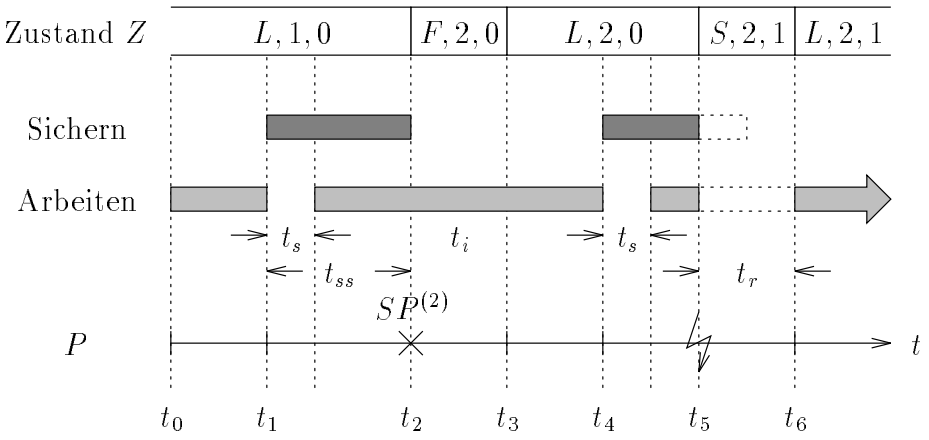

Abbildung 6.1: Zustände eines Prozesses

Zum Zeitpunkt t_0 sei der Prozeß P_i im Zustand $Z = (L, 1, 0)$. Mit dem Abschluß des Arbeitsintervalls wird zum Zeitpunkt t_1 die nebenläufige Zustandssicherung angestoßen. Dadurch wird die Bearbeitung des nächsten Arbeitsintervalls um den Mehraufwand t_s verlängert. Nach Ablauf der tatsächlichen Sicherungszeit t_{ss} ist ein neuer, vorläufiger Sicherungspunkt angelegt. Die Sicherungspunktnummer wird dabei um eins erhöht. Damit kann zum Zeitpunkt t_2 die Freigabephase $(F, 2, 0)$ beginnen. Nach der Freigabe des Sicherungspunkts zum Zeitpunkt t_3 betritt der Prozeß P wieder die Leerphase, der neue Zustand ist $(L, 2, 0)$.

Nach dem Beenden des nächsten Arbeitsabschnitts (Zeitpunkt t_4) wird wiederum die Zustandssicherung gestartet. Allerdings wird dieser Vorgang unterbrochen, da zum Zeitpunkt t_5 ein Fehler erkannt wird. Der neue vorläufige Sicherungspunkt ist noch nicht erstellt, also kann sofort die Synchronisationsphase betreten werden. Dabei wird die Synchronisationsnummer erhöht (Zustand $Z = (S, 2, 1)$). Die Synchronisationsphase wird zum Zeitpunkt t_6 verlassen und das Programm vom letzten Sicherungspunkt $SP^{(2)}$ wieder fortgesetzt ($Z = (L, 2, 1)$).

6.1.2 Zustandsübergänge

Die Situationen, die eine Zustandsveränderung bewirken, sind durch Regeln definiert. Ob eine Regel angewendet werden kann, entscheidet im dezentralen Ansatz jeder Teilnehmer basierend auf seinen Informationen über den Zustand der anderen Teilnehmer. Es sei angenommen, daß diese Information nicht von Fehlern verändert wird. Die Übertragung der Zustandsinformation durch Rundsprüche ist Gegenstand des Abschnitts 6.1.3.

Ist $P = \{P_1, \ldots, P_m\}$ eine Menge von kooperierenden Prozessen, so sei nun der Prozeß $P_j \in P$ mit dem Zustand $Z_j = (p_j, c_j, s_j)$ betrachtet. Jeder Prozeß P_j speichert die jeweils neueste Sicherungspunktnummer und Synchronisationsnummer aller beteiligten Prozesse in den Variablen c_i und s_i. Dabei ist diese Information nur für den Fall $i = j$ tatsächlich aktuell, für alle anderen Prozesse $P_{i,i\neq j}$ repräsentieren c_i und s_i und die Information die P_j zuletzt von dem betreffenden Prozeß P_i empfangen hat. Die Kenntnis der jeweiligen Phase, in der sich die anderen Teilnehmer befinden, ist für die Entscheidung nicht erforderlich.

Fünf Regeln beschreiben die erlaubten Zustandsübergänge des Prozesses P_j. Diese Regeln sollen sowohl die nebenläufige Zweiphasenfreigabe (Abschnitt 5.2), als auch das Konzept des verzögerten Wiederanlaufs (Abschnitt 5.4) umsetzen. Bei einem Fehler muß dazu, falls von dem betreffenden Prozeß bereits ein neuer Sicherungspunkt erstellt wurde, mit dem Wiederanlauf (mit dem Betreten der Synchronisationsphase) solange gewartet werden, bis eine Bedingung vorliegt, die das Verwerfen oder Freigeben des Sicherungspunkts bewirkt. Im Fehlerfall wird also zunächst in der selben Phase wie vor dem Fehler fortgefahren. Wann zum Rücksetzen der Applikation die Synchronisation betreten werden kann, ist durch die Regeln bestimmt. Ist ein Fehler aufgetreten, so sei angenommen, daß mindestens ein Prozeß existiert, der den Fehler selbst erkannt oder der von ihm erfahren hat. Dieser Umstand wird im folgenden dadurch beschrieben, daß in einem solchen Prozeß P_j die Boolesche Variable err wahr ist.

Die Regel (1) definiert den Übergang von der Leerphase in die Freigabephase, Regeln (3) und (4) definieren die Übergänge von der Freigabephase in die Leerphase bzw. in die Synchronisationsphase, Regel (2) definiert den Übergang von der Leerphase zur Synchronisationsphase, und schließlich die Regel (5) den Übergang von

der Synchronisationsphase zur Leerphase. Andere Übergänge sind ausgeschlossen. Das Ausführen einer Regel sei eine unteilbare Aktion eines Teilnehmers.

(1) (Zustandssicherung)
Falls in der Leerphase $Z_j = (L, c_j, s_j)$ der vorläufige Sicherungspunkt $SP^{(c_j+1)}$ lokal erfolgreich gesichert wurde, so inkrementiere die Sicherungspunktnummer und betrete die Freigabephase $Z_j = (F, c_j + 1, s_j)$.

(2) (Fehlererkennung)
Falls ein Teilnehmer in der Leerphase $Z_j = (L, c_j, s_j)$ einen Fehler erkennt oder ein anderer Teilnehmer die Synchronisationsphase betreten hat, d.h.

$$\exists P_i \in P : \quad err \vee s_i > s_j,$$

so erhöhe den Synchronisationszähler s_j, setze err falsch und betrete die Synchronisationsphase $Z_j = (S, c_j, s_j + 1)$.

(3) (Commit)
Falls in der Freigabephase $Z_j = (F, c_j, s_j)$ alle anderen Teilnehmer ebenfalls den Sicherungspunkt mit der Nummer c_j geschrieben haben, d.h. wenn gilt:

$$\forall P_i \in P : \quad c_i \geq c_j,$$

so führe eine Commit-Operation aus und beende die Freigabephase ($Z_j = (L, c_j, s_j)$).

(4) (Abort)
Falls in der Freigabephase $Z_j = (F, c_j, s_j)$ für den Sicherungspunkt $SP^{(c_j)}$ einer der Teilnehmer die Synchronisationsphase betreten hat und dieser Teilnehmer vorher den Sicherungspunkt mit Nummer c_j nicht gesichert hatte, d.h.

$$\exists P_i \in P : \quad s_i > s_j \wedge c_i < c_j,$$

so führe eine Abort-Operation aus, dekrementiere den Sicherungspunktzähler, erhöhe den Synchronisationszähler, setze err falsch und betrete die Synchronisationsphase $Z_j = (S, c_j - 1, s_j + 1)$.

(5) (Synchronisation)
Falls in der Synchronisationsphase $Z_j = (S, c_j, s_j)$ alle Teilnehmer auf den Fehler reagiert haben, d.h.

$$\forall P_i \in P : \quad s_i \geq s_j,$$

so beende die Synchronisationsphase (gehe nach $Z_j = (L, c_j, s_j)$).

Abbildung 6.2 gibt eine Übersicht aller erlaubten Zustandsübergänge.

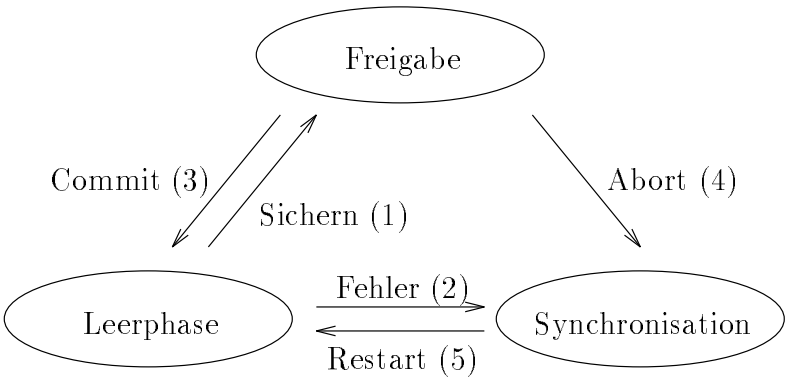

Abbildung 6.2: Zustände und Zustandsübergänge

6.1.3 Nachrichtenübertragung

Die Zustandsnachrichten (Rundsprüche) enthalten die Kennung (Signatur) des Absenders, dessen Sicherungspunktnummer und dessen Synchronisationsnummer. Beschreibt $P = \{P_1, \ldots, P_m\}$ die Menge aller Teilnehmer, so kann als Kennung eines Prozesses P_i der Index i verwendet werden. Eine Nachricht n ist damit ein Tripel dreier natürlicher Zahlen (i, c_n, s_n). Es ist sinnvoll, Nachrichten immer bei solchen Zustandsübergängen zu senden, die eine Veränderung des Synchronisationszählers bzw. Sicherungspunktzählers bewirken, also bei Ausführung der Regeln (1), (2) und (4). Im Beispiel aus Abbildung 6.1 (Seite 87) würde also zu den Zeitpunkten t_2 und t_5 (entsprechend einer `READY-` bzw. einer `SYNC`-Nachricht) eine Zustandsnachricht gesendet werden.

Bei Regeln (1,2) werden entweder die Sicherungspunktnummer oder die Synchronisationsnummer erhöht. Ein Dekrementieren tritt nur bei einer Abort-Operation (Regel (4)) auf, da der vorläufige Sicherungspunkt gelöscht wird. Gleichzeitig wird bei einer Abort-Operation aber auch die Synchronisationsnummer erhöht. Wenn der Synchronisationszähler stärker gewichtet wird, kann die gewichtete Summe der Zählerstände immer nur streng monoton steigen. Diese Eigenschaft kann zum Ordnen der Nachrichten verwendet werden.

Beim Empfangen einer Nachricht $n = (i, c_n, s_n)$ verfährt der Teilnehmer P_j nach der folgenden Aktualitätsregel:

> Gilt
>
> $$s_i > s_n \vee (s_i = s_n \wedge c_i \geq c_n),$$
>
> so ignoriere diese Nachricht, ansonsten setze $s_i = s_n$ und $c_i = c_n$.

Im Beispiel Abbildung 6.1 (Seite 87) möge ein anderer Prozeß die Nachricht $n = (i, 2, 1)$ erhalten, ohne vorher die Nachricht $n = (i, 2, 0)$ empfangen zu haben. Bei

der Prüfung der Aktualitätsbedingung wird sich herausstellen, daß gilt $s_i \not> 1$. Also wird sowohl c_i und s_i aktualisiert. Trifft später noch die Nachricht $n = (i, 2, 0)$ ein, so wird sie ignoriert. Gegenüber dem vorher gespeicherten Zustand $s_i = 0$ und $c_i = 1$ hat sich sowohl die Sicherungspunktnummer und die Synchronisationsnummer verändert, so daß sich für den Empfänger die Frage stellt, ob der Sicherungspunkt vor oder nach dem Wiederanlauf geschrieben wurde. Diese Frage kann aber immer aus dem Kontext (aus der Phase), in dem die Nachricht empfangen wird, eindeutig beantwortet werden. Diese Eigenschaft wird im Abschnitt 6.1.4 nachgewiesen.

Die Bedeutung dieser Eigenschaft ist, anders formuliert, daß das erfolgreiche Empfangen irgendeiner Nachricht alle vorherigen Verluste von Nachrichten desselben Absenders maskiert. Nachrichtenverluste können deshalb durch sehr einfache Mechanismen behandelt werden: Der Sender eines Rundspruchs muß nicht für den Fall einer Rückfrage ältere Nachrichten aufbewahren, da sie durch das Übertragen des aktuellen Zustands ersetzt werden können. Vorteilhaft ist weiterhin, daß seitens des Empfängers im Fall einer Nachrichtenüberholung nicht auf eventuell fehlende Nachrichten gewartet werden muß. Im Vergleich zur Numerierung der Nachrichten entsteht kein zusätzlicher Aufwand, außer daß die Nummer nun zweigeteilt ist. Der Wertebereich des Sicherungspunkt- und des Synchronisationszählers kann darüberhinaus auf nur je zwei Bit beschränkt werden (siehe Abschnitt 6.1.5).

Als Beispiel für die Arbeitsweise des Verfahrens sei nochmals Abbildung 5.3a (Seite 74) betrachtet. Es sei angenommen, das zunächst alle drei Prozessoren sich im selben Zustand befinden, in der Leerphase mit dem Synchronisationszähler s und der Sicherungspunktnummer c. Nach dem Erzeugen des Sicherungspunktes $SP^{(c+1)}$ senden die Prozessoren nach Regel (1) nun, einer `READY`-Nachricht entsprechend, die Zustandsinformation $(i, c + 1, s)$ und betreten die Freigabephase. Nach dem Empfangen der Nachrichten von P_1 und P_3, die nach der Aktualitätsregel neue Information darstellen, wird P_2 den Sicherungspunkt nach Regel (3) für gültig erklären und wieder die Leerphase betreten. P_1 verfährt ebenso. Beim Erkennen des Fehlers sendet P_2 die Zustandsnachricht $(2, c + 1, s + 1)$ und betritt die Synchronisationsphase (Regel (2)). Obwohl P_3 den Sicherungspunkt noch nicht für gültig erklärt hat, kann P_3 dies sofort tun, da die Bedingungen für Regel (3) erfüllt sind. Erst danach betritt P_3 die Synchronisationsphase ebenfalls nach Regel (2). Wenn die verzögerte Zustandsnachricht ankommt, wird sie nach der Aktualitätsregel ignoriert.

Das Beispiel Abbildung 5.3b verdeutlicht nochmals die Bedingung für Regel (4). Nach dem Fehler kann P_2 noch nicht die Freigabephase verlassen, da noch keine Nachricht von P_3 über $SP^{(c+1)}$ vorliegt. Ebenso kann P_1 den Sicherungspunkt $SP^{(c+1)}$ nicht löschen. Erst wenn P_3 auch den Sicherungspunkt $SP^{(c+1)}$ geschrieben hat, werden alle Prozesse die Freigabe ausführen und zum Wiederanlauf die Synchronisationsphase betreten.

6.1.4 Nachweis der Korrektheit

Für das korrekte Arbeiten eines Protokolls zur Rückwärtsfehlerbehebung sind die folgenden zwei Eigenschaften zu erfüllen:

- Das Verfahren ist verklemmungsfrei.
- Beim Wiederanlauf bleibt der Systemzustand konsistent.

Für die Konsistenz des Systemzustands ist es erforderlich, daß alle Prozesse nach einem Wiederanlauf die Leerphase immer mit derselben Sicherungspunktnummer betreten, da nur dann alle Prozesse auf einen Sicherungspunkt mit der selben Nummer zurücksetzen. Zur Rücksetztrennung ist Gleichheit der Synchronisationszähler nötig.

Ausgehend von einem Zeitpunkt, in dem sich alle Prozesse in einer Leerphase $Z = (L, c, s)$ befinden oder diese Phase zwingend betreten werden, kann gezeigt werden, daß alle Prozesse innerhalb endlicher Zeit wieder eine Leerphase $Z = (L, c', s')$ betreten, für die wiederum alle Prozesse gleiche Zählerstände besitzen. Entsprechend den Regeln sind nur drei verschiedene Folgen von Zustandsübergängen (bzw. Zyklen) möglich, bis ein Prozeß P_j die Leerphase erstmalig wieder betritt (vgl. Abbildung 6.2):

(1) Unter der Voraussetzung, daß *kein* Prozeß P_i existiert, für den in der Leerphase $Z = (L, c, s)$ eine Fehlerbedingung *err* auftritt, durchlaufen alle Prozesse die folgenden Zustände:

$$L, c, s \longmapsto^{(1)} F, c+1, s \longmapsto^{(3)} L, c+1, s$$

Das Erstellen eines neuen Sicherungspunkts $SP^{(c+1)}$ hängt von zwei Vorbedingungen ab: Ein Arbeitabschnitt muß abgeschlossen sein und ein freier Sicherungspuffer muß existieren. In einem korrekten Anwendungsprogramm aber kann ein Arbeitsabschnitt immer beendet werden. In der Leerphase steht immer ein freier Puffer zur Verfügung.

Zur Freigabe eines Sicherungspunkts durch den Prozeß P_j ist es notwendig, daß auch alle anderen Prozesse den Sicherungspunkt erstellt und die Freigabephase betreten haben, denn nur dann wurde die Sicherungspunktnummer aller Prozesse inkrementiert. Da für ein korrektes Anwendungsprogramm alle Prozesse den Übergang $L, c, s \longmapsto^{(1)} F, c+1, s$ durchführen können, ist diese Bedingung irgendwann erfüllt, sofern, wie vorausgesetzt, in keinem Prozeß vor dem Sichern von $SP^{(c+1)}$ ein Fehler auftritt.

Sobald der Prozeß P_j die Freigabe ($F, c+1, s \longmapsto^{(3)} L, c+1, s$) durchführen kann, haben auch alle anderen Prozesse die Freigabephase betreten. Das Nachrichtensystem stellt allen anderen Prozessen, wie beschrieben, innerhalb endlicher Zeit die gleiche oder noch aktuellere Zustandsinformation zur Verfügung.

Alle Nachrichten, die aber nach der Aktualitätsregel akzeptiert werden, können in keinem Prozeß einen anderen Übergang auslösen. Also werden alle Prozesse die Leerphase $Z = (L, c+1, s)$ nach Regel (3) betreten.

(2) Unter der Voraussetzung, daß mindestens ein Prozeß P_i existiert, für den in der Leerphase $Z = (L, c, s)$ eine Fehlerbedingung err vorliegt, so geht mindestens ein Prozeß nach der Regel (2) in den Zustand $Z = (S, c, s+1)$. Daraufhin durchlaufen alle übrigen Prozesse eine der beiden Zustandsfolgen:

(a) $L, c, s \longmapsto^{(1)} F, c+1, s \longmapsto^{(4)} S, c, s+1 \longmapsto^{(5)} L, c, s+1$

(b) $L, c, s \longmapsto^{(2)} S, c, s+1 \longmapsto^{(5)} L, c, s+1$

Wie in Fall 1 ausgeführt, kann ein Prozeß in der Leerphase einen neuen Sicherungspunkt erstellen ($L, c, s \longmapsto^{(1)} F, c+1, s$). Hat inzwischen ein anderer Prozeß, wie vorausgesetzt, die Synchronisationsphase betreten, so kann ein Prozeß in der Freigabephase in die Synchronisationsphase übergehen und den vorläufigen Sicherungspunkt löschen ($F, c+1, s \longmapsto^{(4)} S, c, s+1$).

Falls ein Prozeß noch in der Leerphase erkennt, daß ein anderer Prozeß die Synchronisationsphase bereits betreten hat, so kann der Prozeß sofort in die Synchronisationsphase übergehen ($L, c, s \longmapsto^{(2)} S, c, s+1$).

Ein Prozeß in der Synchronisationsphase kann diese Phase nicht verlassen, ohne daß alle anderen Prozesse ebenfalls bereits in der Synchronisationsphase stehen. Bis dieser Fall eingetreten ist, kann ein Prozeß seinen Zustand $Z = (S, c, s+1)$ nicht verändern. Da also mindestens ein Prozeß den Synchronisationszähler $s+1$ besitzt, und dieser Zählerstand innerhalb endlicher Zeit, durch das Nachrichtensystem verbreitet, irgendwann alle anderen Prozesse erreicht, sind auch die anderen Prozesse gezwungen, ebenfalls die Synchronisationsphase nach einer der beiden Möglichkeiten (a) oder (b) zu betreten.

Sobald der Prozeß P_j die Synchronisation ($S, c, s+1 \longmapsto^{(5)} L, c, s+1$) durchführen kann, haben auch alle anderen Prozesse die Synchronisationsphase betreten. Wiederum können alle Nachrichten, die nach der Aktualitätsregel akzeptiert werden, in keinem Prozeß einen anderen Übergang auslösen. Also werden alle Prozesse die Leerphase $Z = (L, c, s+1)$ nach Regel (5) betreten.

Nach der Initialisierung befinden sich alle Prozesse im Zustand $Z = (L, 0, 0)$. Durch Induktion folgt, daß das Verfahren nicht nur in jedem Zyklus, sondern immer die beiden Bedingungen (verklemmungsfrei und konsistent) erfüllt.

6.1.5 Beschränkung des Zählerwertebereichs

In der bisher beschriebenen Form bringt dieses Verfahren einen relativ hohen Speicheraufwand mit sich, denn immerhin muß jeder Teilnehmer die Zählerstände aller

anderen Teilnehmer speichern. Zum einen kann aber die Anzahl der zu speichernden Variablen durch ein hierarchisches Kommunikationsmodell auf $O(log\, n)$ reduziert werden (siehe Abschnitt 6.3), zum anderen kann bei entsprechender Implementierung des Nachrichtenmechanismus der Wertebereich der Zähler auf jeweils zwei Bit beschränkt werden.

Bei Betrachtung des gesamten Systems zu einem beliebigen Zeitpunkt gibt es mindestens einen Prozeß P_j, für dessen Sicherungspunktzähler c_j und Synchronisationszähler s_j gilt:

$$\forall P_i \in P : s_j < s_i \vee (s_j = s_i \wedge c_j \leq c_i)$$

Dieser Prozeß wird im folgenden als der *langsamste* Prozeß bezeichnet, $s = s_j$ heißt die kleinste Synchronisationsnummer und $c = c_j$ heißt die kleinste Sicherungspunktnummer.

Da jeder der drei möglichen Zyklen von Leerphase zu Leerphase eine Synchronisation enthält, gilt zu jedem Zeitpunkt in einem System mit nur zwei Sicherungspunkt-Puffern, orientiert am langsamsten Prozeß P_j, die folgende Bedingung:

$$\begin{aligned} \forall P_i \in P : \quad & s_i = s \wedge c_i = c \\ \text{oder} \quad & s_i = s + 1 \wedge c_i = c \\ \text{oder} \quad & s_i = s \wedge c_i = c + 1 \end{aligned}$$

Für die Sicherungspunktnummer einer Nachricht $n = (i, c_n, s_n)$ gilt beispielsweise:

- Die Nummer $c_n = c - 1$ oder kleiner beschreibt Nachrichten, die veraltet sind, da sie den vorherigen Sicherungspunkt betreffen, dessen Freigabephase schon abgeschlossen ist.
- Eine Nachricht mit der Nummer $c_n = c$ betrifft den aktuellen Sicherungspunkt.
- Die Nummer $c_n = c + 1$ kennzeichnet Nachrichten über den nächsten Sicherungspunkt.

Ab dem Moment, in dem der langsamste Prozeß die Sicherungspunktnummer erhöht, werden keine Nachrichten mehr mit kleinerer Sicherungspunktnummer erzeugt. Kann sichergestellt werden, daß von diesem Moment ab alle Nachrichten mit kleinerer Sicherungspunktnummer vom Kommunikationssystem unterdrückt werden, so reicht es aus, die Sicherungspunktnummern modulo 3 zu zählen. Im folgenden seien dazu die binären Operatoren $-^3$ und $+^3$ wie folgt definiert:

$$\begin{aligned} a +^3 b &= (a + b)\, mod\, 3 \\ a -^3 b &= (a - b)\, mod\, 3 \end{aligned}$$

Die Nummer c beschreibt aus der Sicht des langsamsten Prozesses den aktuellen Sicherungspunkt. Die Nummer $c +^3 1$ kennzeichnet den vorläufigen Sicherungspunkt der weiter fortgeschrittenen Prozesse. Nachrichten mit der dritten Nummer $c -^3 1$, sind veraltet und werden vernichtet. Die Zählweise zeigt Abbildung 6.3. Werden im Sicherungsspeicher mehr als zwei Puffer für die Sicherungspunkte vorgesehen, so muß der Wertebereich des Zählers für die Sicherungspunktnummer entsprechend erhöht werden.

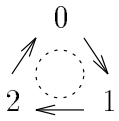

Abbildung 6.3: Zyklische Zählweise für die Sicherungspunktnummern

Eine ähnliche Überlegung kann man für den Synchronisationszähler anstellen.

Bei der Implementierung im MEMSY-Rechner wurde die Beschränkung der Zählerwerte verwendet (Abschnitt 6.2.2).

6.2 Umsetzung im MEMSY-Multiprozessor

Zur programmgesteuerten Zustandssicherung und Rückwärtsfehlerbehebung werden im MEMSY-Rechner die beschriebenen Protokolle eingesetzt[1]. Dabei wurden möglichst nur Systemaufrufe und Schnittstellen der POSIX-Norm [IEE90] verwendet, so daß das Verfahren auf anderen Rechnern einfach umgesetzt werden kann. In [Böh94] wurde beispielsweise eine Umsetzung für SUPRENUM durchgeführt.

6.2.1 Fehlererkennung

Die erweiterten Fehlererkennungsmethoden wurden so entworfen, daß ein Fehler eine Ausnahme auslöst, die das Betriebssystem dem betroffenen Prozeß als ein Signal zustellt. Durch dieses Signal wird der Prozeß normalerweise terminiert. Durch das Installieren einer Behandlungsroutine für das Fehlersignal hat ein Prozeß die

[1]Im Rahmen des MEMSY-Projekts beschäftigen sich weitere Arbeiten mit Fehlertoleranzaspekten. Außer den bereits erwähnten Arbeiten zur Fehlererkennung (Watchdogprozessoren und Master-Checker-Betrieb) wurden in der Koppelhardware fehlermaskierende Ansätze verwirklicht: Im Koppelmodul (siehe Abschnitt 2.1.2) kann durch Zugriff auf eine andere Adresse ein alternativer Weg zu einem Kommunikationsspeicher ausgewählt werden. Zum Abspeichern von Sicherungspunkten ist die Entwicklung eines stabilen RAM-Speichers vorgesehen. Eine Übersicht über das Fehlertoleranzkonzept findet sich in ausführlicherer Form in [DGH+93].

Gelegenheit, das Auftreten eines temporären Fehlers den anderen Prozessen durch eine Fehlermeldung mitzuteilen.

Das Erkennen von dauerhaften Fehlern stellt im MEMSY-Multiprozessor derzeit noch ein ungelöstes Problem dar. Die Vernetzung der Watchdogprozessoren [Mic92] wird möglicherweise zu einem späteren Zeitpunkt die Grundlage für ein Diagnosenetzwerk bilden, in dem die Überwachungshardware unabhängig von der Funktionsbereitschaft der zugeordneten Knoten miteinander kommunizieren kann. Zum gegenwärtigen Zeitpunkt sind allerdings noch keine derartigen Funktionen implementiert. Als Notbehelf wurde in der Pilotimplementierung für das Erkennen von dauerhaften Fehlern die Zeitschrankenüberwachung von Lebenszeichen verwendet. Dabei wurde versucht, diese Lebenszeichen nur über möglichst kurze Kommunikationsstrecken zu versenden, um die Zeitschranken besser abschätzen zu können. Dieses Vorgehen ist in das Kommunikationssystem integriert.

Die Kontrollflußüberwachung, sowohl durch Software (siehe Kapitel 4), als auch durch Hardware (mittels Watchdogprozessor), erfordert eine spezielle Vorverarbeitung der Programme. Durch einen Präprozessor wird das Programm modifiziert, und im Falle der hardwarebasierten Überwachung eine Datenbasis für den Watchdogprozessor erstellt. In der gegenwärtigen Implementierung greift die Kontrollflußüberwachung erst ab der Prozeßebene ein, das Betriebssystem selbst wird nicht überwacht. Die Quelltexte der Bibliotheken, die zu den Programmen dazugebunden werden, können mit dem Präprozessor behandelt werden.

Spezielle Aufmerksamkeit erfordert aber die Kontrollflußüberwachung mit einem Watchdogprozessor:

- Die zum Watchdogprozessor übertragenen Signaturen werden in einem FIFO gepuffert, so daß Fehler erst mit einer gewissen Verzögerung nach dem Entleeren des FIFO erkannt werden.
- Der Watchdog selbst ist zustandsbehaftet. Der Zustand des Watchdogprozessors muß im Sicherungspunkt abgelegt werden und beim Wiederanlauf restauriert werden.

Für die Kontrollflußselbstüberwachung bestehen diese Probleme nicht, da die Signaturen sofort überprüft werden und die Überwachung keinen selbstständigen Zustand besitzt.

Da in der programmgesteuerten Zustandssicherung der Prozeß nach einem Fehler im allgemeinen neu gestartet wird (siehe Abschnitt 5.1), ist das zweite Problem hauptsächlich für die transparente Zustandssicherung von Bedeutung. In [Mic92] wird als Lösung die folgende Vorgehensweise vorgeschlagen: Nach dem Erzeugen eines vorläufigen Sicherungspunkts sendet ein Prozeß eine spezielle `CREATE_CHECKPOINT`-Signatur an den Watchdogprozessor. Stößt der Watchdogprozessor in seinem

FIFO auf diese Signatur, so legt er einen internen Sicherungspunkt an und sendet eine Bestätigung. Erhält der Prozeß die Bestätigung des Watchdogprozessor, so weiß er, daß der Programmlauf bisher fehlerfrei war und daß auch der Watchdogprozessor einen Sicherungspunkt erzeugt hat. Im SEIS-Watchdogprozessor ([PMHH93] werden Zustandssicherung und Wiederanlauf zur Beschleunigung bereits von der Hardware unterstützt.

Für die programmgesteuerte Zustandssicherung reicht es aus, den Watchdogprozessor durch eine spezielle Signatur mit dem Prozeß zu synchronisieren, bevor ein Sicherungspunkt als korrekt gelten kann.

6.2.2 Übertragung der Zustandsnachrichten

Im MEMSY-Rechner sind zwei Rundspruchverfahren möglich, die den in Abschnitt 6.1.3 vorgestellten Ansatz einfach und mit nur geringem Aufwand realisieren. Die Beschränkung des Zählerwertebereichs (Abschnitt 6.1.5) erfordert im Verfahren eine Möglichkeit, veraltete Nachrichten zu vernichten. In beiden Implementierungen kann diese Vorbedingung garantiert werden: Nachrichten werden nur empfangsseitig gepuffert. Beide Verfahren entleeren den Empfangspuffer jeweils bei der atomaren Bearbeitung einer Regel, die die Änderung eines Zählers bewirkt. Beispielsweise werden beim Ausführen von Regel (1), die ein Erhöhen der Sicherungspunktnummer bewirkt, alle gepufferten Nachrichten mit der vorherigen Nummer vernichtet. Daher sind, sobald der langsamste Prozeß die Nummer c erreicht hat, keine Nachrichten mehr mit $c_n = c - 1$ im System.

6.2.2.1 Lokales Netzwerk

Die MEMSY-Rechnerknoten sind mit Anschlüssen für lokale Netze ausgestattet (FDDI und Ethernet). Ein Rundspruch belegt auf solchen Netzen die gleiche Bandbreite wie eine Punkt-zu-Punkt Übertragung. Zum Realisieren einer zuverlässigen Übertragung wird, nach einer einstellbaren Verzögerung, der jeweils aktuelle Zustand neu übertragen. Sollte inzwischen eine Zustandsänderung aufgetreten sein, kann jede neue Nachricht ältere, verlorene Nachrichten ersetzen. Bei permanentem Ausfall des Netzes blockiert das Verfahren.

Die redundante Nachrichtenwiederholung kann zum Erkennen dauerhafter Fehler als Lebenszeichen verwendet werden. Bleibt das Lebenszeichen eines Prozessors mehrmals aus, wird die Programmausführung abgebrochen und dem Bedienpersonal ein dauerhafter Ausfall signalisiert. Da dauerhafte Fehler und Nachrichtenverluste nicht die Regel sind, kann der Abstand zwischen den Lebenszeichen relativ großzügig gewählt werden, ohne daß ein merklicher Einfluß auf die Verfügbarkeit entsteht. Die Belastung des Netzes durch die Lebenszeichen kann damit über einen gewissen Bereich skaliert werden.

Abbildung 6.4: Nachrichtendiffusion

In diesem Beispiel müssen mindestens vier Nachrichtenverluste nahezu gleichzeitig auftreten, damit eine Nachricht verloren geht. Da dies vergleichsweise unwahrscheinlich ist, kann die Nachrichtenübermittlung sorglos und auf Geschwindigkeit optimiert vorgenommen werden. Dennoch kann man davon ausgehen, daß jede Nachricht jeden anderen Prozeß erreicht. In gewissen Grenzen, in Abhängigkeit von der Topologie, ist diese Redundanz skalierbar. Der Übertragungsmechanismus hat einige Vorteile: Die Nachrichten werden immer nur über eine Stufe verschickt. Aufwendige Protokolle für adaptives Routing zum Tolerieren permanenter Verbindungsausfälle sind nicht notwendig. Ein Nachteil ist es, daß im ungünstigsten Fall

[2]In der Realität besitzt der MEMSY-Rechner keine homogene Torus-Struktur, darüberhinaus ist jeder Knoten selbst mit 4 Prozessoren ausgestattet. Im Detail sind deshalb Veränderungen des Verfahrens nötig, die aber nicht prinzipieller Natur sind.

insgesamt $(3(n-1)+4)n = 3n^2+n$ Nachrichten verschickt werden. Allerdings werden dazu $2n$ Verbindungen parallel gleichmäßig ausgenutzt, jede Verbindung wird also durchschnittlich mit maximal $1{,}5n$ Übertragungen belastet.

Auch dieser Nachrichtenmechanismus kann zum Erkennen von dauerhaften Fehlern verwendet werden. Ein Prozeß überträgt an seine Nachbarn in regelmäßigen Abständen seinen aktuellen Zustand als Lebenszeichen. Da diese Lebenszeichen meistens nicht die Aktualitätsbedingung erfüllen, werden sie nicht weitergesendet. Im Unterschied zur Übertragung im lokalen Netz werden die Lebenszeichen nur lokal überwacht.

Im Vergleich zu einem zentralen Ansatz werden alle Verbindungen gleichmäßig belastet. Es entstehen keine lokalen Mehrbelastungen wie beispielsweise in den Kommunikationspfaden eines zentralen Koordinators.

6.2.3 Zustandssicherung und Wiederanlauf

Als Sicherungsspeicher wurde in der Implementierung die lokale Festplatte der Rechnerknoten verwendet. Die Anpassung anderer Speichermedien sollte kein Problem darstellen, sofern die Ansteuerung dem UNIX-Gerätetreiberkonzept entspricht.

Zur nebenläufigen Zweiphasen-Freigabe werden die Koordinierungsnachrichten asynchron empfangen. Bei der Netzwerkimplementierung wird beim Empfangen eines Rundspruchs dem Prozeß das `SIGPOLL`-Signal zugestellt. Bei der Nachrichtendiffusion wird der Empfangspuffer nach Ablauf eines Intervalltimers (`SIGALRM`-Signal) durchsucht. Die Protokolle werden jeweils innerhalb der Signal-Behandlungsroutine bearbeitet.

Zur nebenläufigen Sicherung wird mittels dem `fork()`-Aufruf, wie in Abschnitt 5.3.2 angedeutet, ein paralleler Prozeß zum Sichern der Daten erzeugt. Falls eine Pufferung der Sicherungsdaten erwünscht ist, so dient dazu der „Buffer-Cache“ des Systems, dessen Größe bei der Systemgenerierung konfiguriert werden kann.

Der Betriebssystemprozeß, der auf den Knotenrechnern die Applikationen startet, kann so konfiguriert werden, daß er Prozesse nach einem Fehler neu startet. Da jede Applikation als ein Sohnprozeß dieses Prozesses gestartet wird, erhält der Betriebssystemprozeß beim Terminieren einen Rückgabewert, mit dem er erkennen kann, ob der Sohnprozeß durch einen Fehler terminiert wurde. Wenn ja, wird die Applikation neu gestartet, falls nein, wurde die Applikation normal beendet.

Beim Neustart (Wiederanlauf nach einem Fehler) erhält aber die Applikation automatisch eine neue Applikationsnummer zugewiesen. Aus diesem Grund hat die Synchronisationsnummer zur Rücksetztrennung keine Bedeutung, da unter MEMSOS nur Prozesse mit gleicher Applikationsnummer kommunizieren können. Weiterhin wird bereits eine Synchronisation außerhalb der Applikation, vor dem Neustart,

durch die Systemprozesse ausgeführt. Dieses Verhalten vereinfacht die Implementierung des Rücksetzprotokolls, da ein Teil der Funktionalität bereits im Betriebssystem vorhanden ist.

Bei einem Systemneustart, z.B. nach einem Stromausfall, ist aber auch der Systemprozeß von dem Fehler betroffen. Um auch dann das Programm fortsetzen zu können, muß der Prozeß beim Neustart ein Logbuch in einem nichtflüchtigen Speicher vorfinden, in dem verzeichnet ist, welche Programme beim Absturz aktiv waren. Bei dauerhaften Fehlern wird das Programm solange nicht gestartet, bis alle notwendigen Knotenrechner wieder einsatzbereit sind.

Bei einem Softwarefehler der Applikation oder beim Auftreten eines Fehlers, der beim Schreiben des Sicherungspunkts noch verborgen war (latenter Fehler), kann das Programm auch nach mehrfacher Wiederholung nicht fortgesetzt werden. Im Sicherungspunkt wird deshalb jeder Wiederanlaufversuch verzeichnet. Wird eine vorher bestimmte Anzahl von Rücksetzversuchen überschritten, kann nur noch das Bedienpersonal weitere Wiederanlaufversuche starten.

6.2.4 Verwaltung der Zustandsinformation

Die Verwaltung der Zustände und Zustandsübergänge muß für jeden Prozeß P_j drei wichtige Voraussetzungen zu erfüllen:

- Zur Sicherung des Prozeßzustands $Z = (L, c, s)$ über Fehler hinweg ist die Speicherung des Zustands in einem stabilen Speicher notwendig (vgl. Abschnitt 6.1.1).
- In Abschnitt 6.1.2 wurde vorausgesetzt, daß die Zustandsaufzeichnungen über die anderen Prozesse (die Variablen c_i und s_i) ebenfalls Fehler überstehen.
- Die Regeln für die Zustandsübergänge müssen atomar ausgeführt werden (vgl. Abschnitt 6.1.2), da ansonsten ein Fehlverhalten der Protokolle auftreten kann.

Beim ersten Punkt, der Speicherung des Prozeßzustands, sind Vereinfachungen möglich. Zunächst sind je nachdem, ob bei der Fehlererholung Sicherungspunkte im stabilen Speicher vorhanden sind, bereits Rückschlüsse auf den Zustand vor dem Fehler möglich: Ist ein vorläufiger Sicherungspunkt vorhanden, so befand sich der Prozeß vor dem Fehler in der Freigabephase. Außerdem kann die Sicherungspunktnummer festgestellt werden. Durch die externe Synchronisierung (siehe vorheriger Abschnitt) und wegen der Rücksetztrennung mittels unterschiedlicher Applikationsnummern tritt die Bedeutung der Synchronisationsnummer im MEMSY-Rechner in den Hintergrund, so daß außer den Sicherungspunkten und dem Logbuch des Systemprozesses keine weiteren Aufzeichnungen auf einem stabilen Medium notwendig sind.

Auch die Zustandsaufzeichnungen über die anderen Prozesse (zweite Voraussetzung) müssen nicht in einem stabilen Speicher abgelegt werden. Bei der Fehlererholung werden die Variablen c_i und s_i einfach auf feste Werte gesetzt. Diese Anfangswerte der c_i und s_i werden so gewählt, daß keine Zustandsänderung auftreten kann, wenn nicht vorher eine Zustandsübertragung aller anderen Prozesse empfangen wurde. Bei der Fehlererholung sendet jeder Prozeß grundsätzlich eine Zustandsnachricht.

Um die atomare Bearbeitung der Regeln zu garantieren (dritte Voraussetzung) werden die entsprechenden Programmabschnitte innerhalb der Prozesse vor Unterbrechungen geschützt; beispielsweise dürfen während der Regelbearbeitung keine Nachrichten asynchron empfangen werden. Unterbrechungen durch unvorhersehbare Ursachen, z.B. durch Fehler, führen zum Neustart.

Bei der Initialisierung der Zustandsdaten eines Applikationsprozesses (bei der Fehlererholung oder beim Programmbeginn) wird zunächst der Sicherungsspeicher durchsucht. Anschließend werden drei Fälle unterschieden:

(1) Der Prozeß findet einen Sicherungspunkt mit der Nummer c vor. Das Programm wurde durch einen Fehler in der Leer- oder Synchronisationsphase abgebrochen. Der Prozeß startet in der Synchronisationsphase mit dem Zustand $Z = (S, c, 1)$. Alle c_i werden auf c gesetzt, die s_i aller anderen Prozesse auf $s_i = 0$.

(2) Der Prozeß findet zwei Sicherungspunkte $c -^3 1$ und c vor. Der Prozeß wurde in der Freigabephase abgebrochen. Der Prozeß startet in der Freigabephase $Z = (F, c, 0)$. Da ein Wiederanlauf erforderlich ist, wird das Vorliegen einer Fehlerbedingung mittels err wahr angezeigt, so daß nach Abschluß der Freigabephase auch tatsächlich die Synchronisationsphase betreten wird. Die c_i aller anderen Prozesse werden auf $c_i = c -^3 1$ gesetzt, alle s_i auf $s_i = 0$.

(3) Der Prozeß findet keinen Sicherungspunkt vor, d.h. es handelt sich um den Programmanfang. Der Prozeß beginnt in der Leerphase $Z = (L, 0, 0)$. Für alle i wird c_i und s_i auf Null gesetzt.

Die Eigenschaften des Verfahrens garantieren, daß bei jeder Unterbrechung für alle Prozesse immer ein gemeinsamer Sicherungspunkt mit der gleichen Nummer vorhanden ist. Ein Prozeß, der nur einen Sicherungspunkt vorfindet, zwingt mit seinem Zustand $Z = (S, c, 1)$ alle Prozesse in der Freigabephase des Sicherungspunkts $SP^{(c+1)}$, diesen zu verwerfen $(F, c+1, 0 \longmapsto^{(4)} S, c, 1)$. Andererseits werden Prozesse in der Freigabephase für $SP^{(c)}$ den Sicherungspunkt noch freigeben und anschließend ebenfalls die Synchronisation betreten $(F, c, 0 \longmapsto^{(3)} L, c, 0 \longmapsto^{(2)} S, c, 1)$. Nach der Synchronisation wird das Programm im Zustand $Z = (L, c, 1)$ fortgesetzt.

6.3 Kommunikationsaufwand

Da das vorhandene Testsystem mit nur 20 Rechnerknoten relativ klein war, bleibt der Kommunikationsaufwand unter der Grenze für die Meßbarkeit. Entsprechend können die folgenden Ausführungen nicht anhand von Meßwerten belegt werden. Auch bei der Implementierung im SUPRENUM-Rechner [Böh94] mit 128 Prozessoren hat sich herausgestellt, daß die Zeiten für die Nachrichtenübertragung bzw. für die Koordinierungsprotokolle nur einen verschwindenden Anteil am Aufwand für die Zustandssicherung bzw. für den Wiederanlauf haben. Stattdessen soll hier versucht werden, rein rechnerisch zu bestimmen, inwieweit die Kommunikationsbandbreite des jeweiligen Mediums durch die beiden Kommunikationsverfahren in Anspruch genommen wird.

Die bei der Kommunikation über das lokale Netz verwendeten Datagramm-Rundsprüche sind angesichts der Kürze der Nutzinformation in ihrer Länge hauptsächlich durch die Adreß- und Verwaltungsinformation bestimmt. Insgesamt dürfte die Länge eines Pakets etwa 100 Byte nicht überschreiten. Weiter sei angenommen, daß die Bandbreite einer Ethernet-Leitung (10 MByte/s) bis zu maximal 5% ausgenutzt werden kann (verfügbare Nutzbandbreite 0,5 MByte/s). Wenn nicht mehr als 0,1% der Nutzbandbreite belegt werden soll, dürfen maximal etwa fünf Zustandsnachrichten je Sekunde übertragen werden. Die regelmäßigen Lebenszeichen zur Erkennung von dauerhaften Fehlern belegen eine gewisse Netzwerkbandbreite, die aber durch geeignete Zeitintervalle skaliert werden kann. Um also obiges Ziel zu erreichen, dürfen tausend Prozessoren im Abstand von 200 Sekunden ein Lebenszeichen senden. Die Reaktionszeit bei dauerhaften Fehlern, innerhalb der ein Fehler dem Bedienungspersonal angezeigt wird, ist also im Minutenbereich. Nach Abschnitt 3.5 ist damit noch eine ausreichend kleine durchschnittliche Rücksetzzeit möglich. Auch der Aufwand für die Behandlung von Unterbrechungen beim Empfangen der Daten ist gering.

Vorübergehend, beispielsweise beim Betreten der Freigabephase, in der die Prozesse der Applikation einen hohen Grad an Synchronität erreichen, entstehen zeitlich begrenzte Lastspitzen, besonders bei großen Prozessorzahlen. Betreten tausend Prozesse innerhalb von einer Sekunde die Freigabephase, werden bereits 20% der Nutzbandbreite durch die Koordinierung verwendet, womit auch bereits eine relativ hohe CPU-Last in allen Knoten entsteht. Für größere Prozessorzahlen ist diese Art der Übertragung also nicht mehr geeignet.

Beim Nachrichtendiffusionsverfahren werden MEMSOS-Nachrichten verwendet. Die Bandbreite einer MEMSOS-Nachrichtenverbindung zwischen zwei Nachbarn liegt z.Zt. bei etwa 5000 Nachrichten pro Sekunde. In einem Torus-Netzwerk sendet jeder Prozessor an vier Nachbarn ein Lebenszeichen, genauso werden vier Lebenszeichen empfangen. Werden Lebenszeichen im Abstand von etwa 10 Sekunden wiederholt, wird damit weit weniger als 0,1% der Nachrichtenbandbreite belegt. Außer vom Abstand der Lebenszeichen hängt die Reaktionszeit auf dauerhafte Fehler von der Ausbreitungsgeschwindigkeit der Nachrichten und vom Kommunikationsdurch-

messer ab. Die Reaktionszeiten für sehr große Systeme können aber immer ohne besonderen Aufwand ebenfalls im Minutenbereich liegen.

Für die Übertragung aller Nachrichten zur Freigabe eines Sicherungspunkts werden bei 1000 Prozessoren maximal insgesamt 1500 Nachrichten pro Kommunikationsverbindung verschickt, d.h. es können bereits Verzögerungen auftreten, die durch eine geringe Erhöhung der CPU-Last den Sicherungsmehraufwand t_s um ein bis zwei Sekunden erhöhen. Für größere Prozessorzahlen ist eine Verringerung der Kommunikationslast erforderlich.

Eine Reduktion der Nachrichtenzahl ist möglich, wenn das System hierarchisch untergliedert wird. Gruppen von Prozessoren versenden zunächst nur intern Nachrichten. Stellt sich heraus, das eine Prozessorgruppe einen Rechenfortschritt erzielt hat, z.B. wenn alle Teilnehmer in dieser Gruppe einen neuen Sicherungspunkt erstellt haben, so wird dieser Umstand in der nächsthöheren Hierarchiestufe verbreitet. Erst wenn alle in der höchsten Hierarchiestufe enthaltenen Teilnehmergruppen den Fortschritt bestätigt haben, kann die Commit-Operation ausgeführt werden. Die Unterteilung in Gruppen kann für die Topologie eines Systems optimiert werden.

Als Beispiel soll das Erstellen eines Sicherungspunkts in einer Torus- oder Feldstruktur betrachtet werden (Abbildung 6.5).

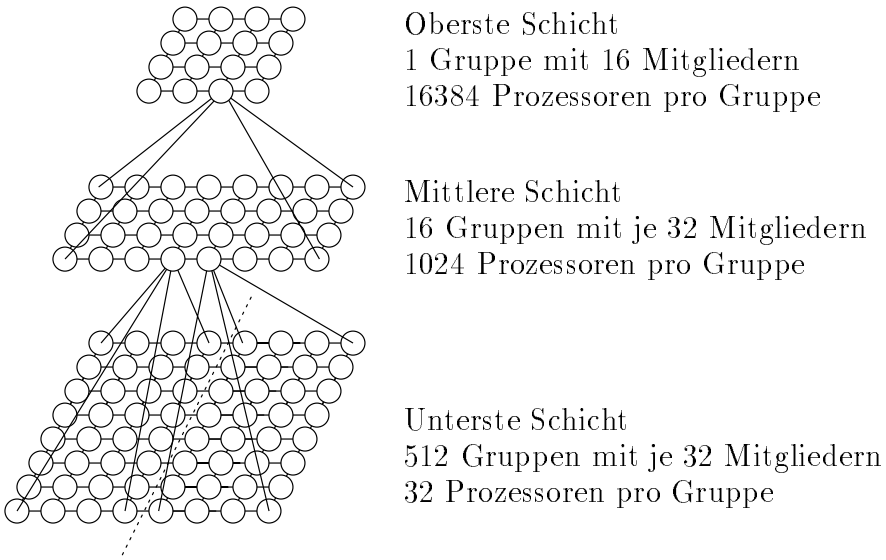

Abbildung 6.5: Hierarchisches Kommunikationsmodell

Insgesamt 16384 Prozessoren wurden in drei Hierarchiestufen eingeteilt, die jeweils $32 \cdot 32 \cdot 16$ Mitglieder besitzen. Innerhalb jeder der untersten Gruppen, jeweils bestehend aus 32 Prozessoren, werden zunächst nach dem Nachrichtendiffusionsverfahren die Zustandsmeldungen aller Mitglieder gesammelt. Hat sich der Sicherungs-

punktzähler der Mitglieder erhöht, so wird der Sicherungspunktzähler der Gruppe inkrementiert und innerhalb der $32 \cdot 32$ Prozessoren einer Gruppe der mittleren Hierarchiestufe verbreitet. Haben wiederum alle 32 Mitglieder einer Gruppe den Sicherungspunktzähler erhöht, wird die Sicherungspunktnummer der mittleren Stufe erhöht. Eine entsprechende Zustandsnachricht wird innerhalb aller $32 \cdot 32 \cdot 16$ Prozessoren des gesamten Systems verbreitet. Sobald alle 16 Sicherungspunktnummern der mittleren Stufe erhöht wurden, kann der Sicherungspunkt freigegeben werden.

Jeder Prozessor muß 32 Nachrichten der untersten Hierarchiestufe, 32 Nachrichten der mittleren, und 16 Nachrichten der obersten Stufe verbreiten. Pro Verbindung entsteht damit eine Last von maximal $80 \cdot 1,5$ Nachrichten. Der Aufwand zur Freigabe eines Sicherungspunktes ist in diesem Beispiel auch für ein System mit 16384 Prozessoren an der Grenze der Meßbarkeit und verschwindet gegenüber der Schreibzeit der Sicherungspunkte. Allgemein reduziert sich durch diese Vorgehensweise die Ordnung des Kommunikationsaufwands pro Verbindung auf $O(log\, n)$. Eine ähnliche Reduzierung des Kommunikationsaufwands wurde auch bei der Implementierung eines hierarchischen Rundspruchverfahrens für den SUPRENUM-Rechner in [Alt93] beobachtet.

Kapitel 7

Zusammenfassung

Die maximal mögliche Parallelitätsgrad in Multiprozessoren ist nicht nur durch die Eigenschaften der numerischen Verfahren, sondern auch durch die Fehlertoleranzeigenschaften bestimmt. In Abhängigkeit von Voraussetzungen und Annahmen, die den heutigen Stand der Technik repräsentieren, wurde gezeigt, daß besonders die Güte der Fehlererkennungsmaßnahmen und, im Rahmen der Rückwärtsfehlerbehebung, die Effizienz der Zustandssicherung einen entscheidenden Einfluß auf die Realisierbarkeit von massiv parallelen Rechnersystemen ausüben. Wegen des bereits ohne Redundanz hohen Hardwareaufwands in solchen Systemen wurden in dieser Arbeit speziell Softwaretechniken vorgestellt. Alle Verfahren wurden u.a. im MEMSY-Multiprozessor implementiert, getestet und bewertet.

Zur Bewertung von Fehlertoleranzmaßnahmen wurden zwei Größen verwendet: Die Verfügbarkeit eines Rechners im Nutzbetrieb sowie die Wahrscheinlichkeit für ein korrektes Rechenergebnis (Vertrauenswürdigkeit). Je nach den Kosten eines fehlerhaften Rechenergebnisses hat die Vertrauenswürdigkeit oder die Minimierung des Leistungsverlusts durch Fehlertoleranzmaßnahmen die höhere Priorität.

Die Verfügbarkeit wird im wesentlichen durch die Effizienz der Zustandssicherung beeinflußt. Weitere, weniger wichtige Faktoren sind Fehlerlatenzzeit und die Erholungszeit nach einem Fehler. Verfahren zur schnellen Rekonfiguration bei dauerhaften Ausfällen haben keinen wesentlichen Einfluß auf die Verfügbarkeit.

Die Vertrauenswürdigkeit der Rechenergebnisse wird im wesentlichen von der Güte der Fehlererkennungsmaßnahmen (Fehlerüberdeckung) beeinflußt. Es zeigt sich, daß beim derzeitigen Stand der Technik durch Investition in Fehlererkennungsmaßnahmen mehr gewonnen werden kann, als durch die Optimierung der Zustandssicherung und Fehlerbehebung.

In anderen Arbeiten wurde bereits nachgewiesen, daß durch Kontrollflußüberwachung einer hoher Anteil der CPU-Fehler erkannt werden kann. In dieser Arbeit wurde eine neuartige Softwaremethode zur Fehlererkennung vorgestellt: die Kontrollflußselbstüberwachung. Diese Methode ist eine Abwandlung des ESIC-

Verfahrens zur Kontrollflußüberwachung mit einem Watchdogprozessor. Vor dem Kompilieren wird der Programmablauf des zu überwachenden Programms von einem Präprozessor analysiert. Anschließend wird noch auf Hochsprachenebene in skalierbaren Abständen Programmcode eingefügt, der den Programmablauf auf Korrektheit überprüft. Mit Fehlerinjektionstests wurde nachgewiesen, daß für eine günstige Einstellung der Parameter des Verfahrens (RF-5) je nach Rechnersystem bei einem Laufzeitmehraufwand um 13–30% eine Verbesserung der Überdeckung von CPU-Fehlern um 20–40% erzielt werden kann. Nachteilig ist der relativ hohe Speichermehraufwand des ausführbaren Programms (etwa 70%). Die mittlere Fehlerlatenz des Verfahrens übertrifft bei weitem die Anforderungen.

Im Bereich der Fehlererkennung mit Softwaremethoden könnte in weiteren Untersuchungen auch daß SEIS-Verfahren zu einem Verfahren der Kontrollflußselbstüberwachung angepaßt werden. Vorteilhaft wäre der geringere Speichermehraufwand. Über den Laufzeitmehraufwand eines solchen Verfahrens kann noch nichts ausgesagt werden, die Fehlererkennungseigenschaften müßten aber die gleichen sein wie bei dem in dieser Arbeit vorgestellten Verfahren.

Die Erhöhung der Fehlerüberdeckung durch Methoden der Kontrollflußüberwachung erscheint aber für die Verwendung in massiv parallelen Rechnersystemen als nicht ausreichend. Das Verwenden von beispielsweise algorithmenspezifischen Überprüfungen oder Hardwaretechniken wie Master-Checker-Betrieb ist zur Ergänzung unbedingt notwendig.

Mittels Rückwärtsfehlerbehebung kann die Verfügbarkeit eines Hochleistungsparallelrechners mit geringem Aufwand deutlich erhöht werden. Für speichergekoppelte Systeme wie MEMSY sind insbesondere Verfahren mit globaler Zustandssicherung geeignet. Dabei erlaubt die programmgesteuerte Zustandssicherung dem Programmierer, selbst den Zeitpunkt und Umfang der Sicherung festzulegen. Neben einer erheblichen Verringerung des Speicherbedarfs eines Sicherungspunkts gegenüber transparenten Verfahren (für typische Anwendungsprogramme auf etwa 15–50%), kann diese Art der Zustandssicherung sehr leicht auf verschiedene Rechnersysteme portiert werden. Zum Vergleich wurde aber auch ein Verfahren zur transparenten Zustandssicherung im MEMSY-Rechner für diese Arbeit implementiert.

Der Rechenzeitaufwand, der für die Zustandssicherung entsteht, ist bei kleineren Systemen hauptsächlich durch die Transferzeit zum Übertragen der Sicherungsdaten auf einen nichtflüchtigen Speicher bestimmt. Erst mit zunehmender Systemgröße gewinnt die Kommunikation der Konsistenzprotokolle an Einfluß. Durch Nebenläufigkeit kann das Einwirken beider Einflüsse erheblich verringert werden: durch den parallelen Ablauf der Sicherungsprotokolle und durch Pufferung der Sicherungsdaten. Anstelle der Pufferung kann durch das Verwenden von Funktionen der virtuellen Speicherverwaltung des Betriebssystems („copy-on-write“) der Zeitbedarf der Zustandssicherung weiter verkleinert werden.

Der parallele Ablauf der Protokolle kann besonders für größere Systeme den Einfluß

des Wartens auf Koordinierungsnachrichten erheblich reduzieren. Für die Umsetzung der Protokolle hat es sich als sinnvoll erwiesen, die Fehlererkennung und die Zustandssicherung im selben Nachrichtenkontext zu betrachten. Anstelle von aufwendigen Übertragungsprotokollen konnten deshalb vergleichsweise einfache Verfahren verwendet werden, die auch in massiv parallelen Systemen effizient umgesetzt werden können. Das in dieser Arbeit vorgestellte Protokoll verwendet einen verteilten Ansatz. Um die Portierbarkeit und Skalierbarkeit des Verfahrens zu verbessern, wurde das Verwenden von Zeitschrankenüberwachung vermieden.

Derzeit sind weitere Untersuchungen in Arbeit, die die Unterstützung der Kommunikationsprotokolle durch ein speziell entwickeltes optisches Bussystem untersuchen.

Bei der Implementierung für den MEMSY-Multiprozessor kann bei verteilter Speicherung der Sicherungsdaten (auf jeweils der lokalen Festplatte eines Knotens) mit einem Mehraufwand von etwa 2 Sekunden je Megabyte Sicherungsdaten gerechnet werden. Mit diesen Werten können mehrere tausend MEMSY-Knoten (sowohl bei transparenter und programmgesteuerter Zustandssicherung) mit relativ geringen Leistungsverlusten betrieben werden. Selbst für sehr große Rechner ist zu erwarten, daß der Leistungsverlust, unter Einsatz der vorgestellten Zustandssicherungsverfahren, auf nur 5–10% begrenzt werden kann.

Betrachtet man die Redundanz, die eingesetzt werden muß, selbst um eine geringe Erhöhung der Fehlerüberdeckung zu bewirken, so zeigt es sich, daß dieser Mehraufwand den möglichen Verlust durch eine ineffiziente Implementierung der Rückwärtsfehlerbehebung weit übersteigt.

Insgesamt ist bei einer Kosten-Nutzen-Analyse von massiv parallelen Rechnern neben der Verschlechterung der numerischen Eigenschaften mit zunehmender Parallelität auch der Effizienzverlust durch die notwendigen Fehlertoleranzmaßnahmen zu berücksichtigen. Insbesondere ist im Einzelfall genau zu prüfen, ob der Einsatz eines Parallelrechners mit hoher Prozessorzahl überhaupt rentabel ist.

Literaturverzeichnis

[AL81] Anderson, T.; Lee, P.A. *Fault Tolerance, Principles and Practice.* Prentice-Hall, 1981.

[Alt93] Altmann, J. Diagnoseprotokolle in Multiprozessorsystemen. Diplomarbeit im Fach Informatik, Universität Erlangen-Nürnberg, Februar 1993.

[And79] Andrews, D. M. Using Executable Assertions for Testing and Fault Tolerance. In *Proc. 9th FTCS*, S. 102–105, 1979.

[Avi85] Avizienis, A. The N-Version Approach to Fault-tolerant Software. *IEEE Transaction on Software Engineering*, SE-11(12), S. 1491–1501, 1985.

[BA84] Brahme, D.; Abraham, J. A. Functional Testing of Microprocessors. *IEEE Transactions on Computers*, C-33(6), S. 475–485, Juni 1984.

[Bac86] Bach, M. J. *The Design of the UNIX Operating System.* Prentice Hall, 1986.

[Bäh91] Bähring, H. *Mikrorechner-Systeme*, S. 555. Springer, 1991.

[BBG83] Borg, A.; Baumbach, J.; Glazer, S. A Message System Supporting Fault Tolerance. In *Proceedings of the ACM Symposium on Operating System Principles*, S. 90–99, Oktober 1983.

[BEG86] Belli, F.; Echtle, K.; Görke, W. Methoden und Modelle der Fehlertoleranz. *Informatik Spektrum*, S. 68–81, April 1986.

[Bel92] Bell, G. Ultracomputer: A Teraflop Before Its Time. *Comm. ACM*, 35(8), S. 27–47, 1992.

[BGH87] Bartlett, J.G.; Gray, J.; Horst, B. Fault Tolerance in Tandem Computer Systems. In Avizienis, A.; Kopetz, H.; Laprie, J. C., Hrsg., *The Evolution of Fault-Tolerant Computing*, S. 55–76. Springer, 1987.

[BM91] Bauch, A.; Maehle, E. Self-Diagnosis, Reconfiguration and Recovery in the Dynamical Reconfigurable Multiprocessor System DAMP. In Dal

Cin, M.; Hohl, W., Hrsg., *Proceedings of the 5th Int. GI/ITG/GMA Conference on Fault Tolerant Computing Systems*, Band 283, *IFB*, S. 18–29. Springer, 1991.

[BMRS91] Banatre, M.; Muller, G.; Rochat, B; Sanchez, P. Design Decisions for the FTM: A General Purpose Fault Tolerant Machine. In *Proc. 21st FTCS*, S. 71–78, 1991.

[Böh94] Böhm, A. *Algorithmenbasierte Fehlertoleranz auf Multiprozessoren: Fehlererkennende Algorithmen in spezialisierter Laufzeitumgebung.* Dissertation, Univ. Erlangen-Nürnberg, 1994.

[BP91] Bowen, N. S.; Pradhan, D. K. A virtual memory translation mechanism to support checkpoint and rollback recovery. In *Proceedings Supercomputing 1991*, S. 890–898, 1991.

[BP92] Bowen, N. S.; Pradhan, D. K. Virtual checkpoints: Architecture and performance. *IEEE Transactions on Computing*, 41(5), S. 516–525, Mai 1992.

[BS83] Barigazzi, G.; Strigini, L. Application-transparent Setting of Recovery Points. In *Proc. 13th FTCS*, S. 48–55, 1983.

[Bur92] Burkhardt, H. Technical Summary of KSR-1. Technical report, Kendall Square Research Corporation, Waltham, MA, 1992.

[CASD85] Cristian, F.; Aghili, H.; Strong, R.; Dolev, D. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. In *Proc. 15th FTCS*, S. 200–206, 1985.

[CJ91] Cristian, F.; Jahanian, F. A Timestamp-Based Checkpointing Protocol for Long-Lived Distributed Computations. In *Proc. 10th Symposium on Reliable Distributed Systems*, S. 12–20, 1991.

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

[Con93] Convex Press, Richardson, Tx. *Exemplar Architecture*, November 1993.

[CS90] Czeck, E. W.; Siewiorek, D. P. Effects of Transient Gate-Level Faults on Program Behavior. In *Proc. 20th FTCS*, S. 236–243, 1990.

[Dal79] Dal Cin, M. *Fehlertolerante Systeme.* Teubner, 1979.

[DGH+93] Dal Cin, M.; Grygier, A.; Hessenauer, H.; Hildebrand, U.; Hönig, J.; Hohl, W.; Michel, E.; Pataricza, A. Fault Tolerance in Distributed Shared-Memory Multiprocessors. In Bode, A.; Dal Cin, M., Hrsg., *Parallel Computer Architectures*, Band 732, *LNCS*, S. 31–48. Springer, 1993.

[DGT86] Dal Cin, M.; Großpietsch, K.; Trautwein, M. Methoden der Fehlerdiagnose. *Informatik Spektrum*, S. 82–94, April 1986.

[DHHP94] Dal Cin, M.; Hohl, W.; Hönig, J.; Pataricza, A. MEMSY — A Modular Expandable Multiprocessorsystem with Fault Tolerance. In *Proc. of the 8th IEEE Int. Parallel Processing Symposium, Cancun, Mexiko*, S. 21–28, April 1994.

[DHMP93] Dal Cin, M.; Hohl, W.; Michel, E.; Pataricza, A. Error detection in massively parallel multiprocessors. In *IEEE Proc. Euromicro Workshop on Parallel and Distributed Processing*, S. 401–408, 1993.

[Ech90] Echtle, K. *Fehlertoleranzverfahren.* Springer, 1990.

[EN91] Echtle, K.; Niedermaier, A. Efficient recovery of statically redundant systems. In *Proceedings of the 5th Int. GI/ITG/GMA Conference on Fault Tolerant Computing Systems*, Band 283, *IFB*, S. 20–41. Springer, 1991.

[ES84] Eifert, J. B.; Shen, J. P. Processor monitoring using asynchronous signatured instruction streams. In *Proc. 14th FTCS*, S. 394–399, 1984.

[Fel88] Fellner, W.-D. *Computergrafik.* Nr. 58 in Reihe Informatik. B.I. Wissenschaftsverlag, 1988.

[Fuj92] Fujitsu America Inc., San Jose, CA. *VPP500 Vector Parallel Processor*, 1992.

[GD78] Gelenbe, E.; Derochette, D. Performance of Rollback-Recovery Systems under Intermittent Failures. *Comm. ACM*, 21(6), S. 493–499, 1978.

[GD94] Guenter, W.; Dal Cin, M. Verteilte Systemdiagnose und Fehlermaskierung. In Wedekind, H., Hrsg., *Verteilte Systeme*, S. 161–176. BI Wissenschaftsverlag, 1994.

[GHPW90] Geist, G.; Heath, M.; Peyton, B.; Worley, P. A Portable Instrumented Communication Library, C Reference Manual. Technical report, ORNL/TM–11133, Oak Ridge National Laboratory, 1990.

[GKP88] Gibson, G.; Katz, R.; Paterson, D. A Case Study for Redundant Arrays of Inexpensive Disks (RAID). In *Proc. SIGMOD Int. Conf. of Management of Data*, S. 109–116, 1988.

[Gör89] Görke, W. *Fehlertolerante Rechensysteme*, Band 2.1, *Handbuch der Informatik.* Oldenbourg, 1989.

[GS91] Geist, G.; Sunderam, V. The PVM System: Supercomputing Level Concurrent Computations on a Heterogeneous Network of Workstations. In *Sixth Distributed Memory Computing Conference Proceedings*, S. 258–261. IEEE Computer Society Press, 1991.

[Hac85] Hackbusch, W. *Multi-Grid Methods and Applications.* Springer, 1985.

[HB85] Hwang, K.; Briggs, F. A. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1985.

[HDG+93] Hofmann, F.; Dal Cin, M.; Grygier, A.; Hessenauer, H.; Hildebrand, U.; Linster, U.; Thiel, T.; Turowski, S. MEMSY - A Modular Expandable Multiprocessor System. In Bode, A.; Dal Cin, M., Hrsg., *Parallel Computer Architectures*, Band 732, *LNCS*. Springer, 1993.

[Hes89] Hessenauer, H. Zuverlässigkeit elementarer Mechanismen zur Realisierung der Synchronisation bzw. des gegenseitigen Ausschlusses. *Arbeitsberichte des IMMD, Univ. Erlangen-Nürnberg*, 22(13), S. 99–111, 1989.

[Hil92] Hildebrand, U. *Konzeption, Bewertung und Realisierung einer dynamischen Netzwerkkomponente für speichergekoppelte Multiprozessorsysteme.* Dissertation, Arbeitsberichte des IMMD, 25(5), Univ. Erlangen-Nürnberg, 1992.

[HMW85] Händler, W.; Maehle, E.; Wirl, K. The DIRMU Testbed For High Performance Multiprocessor Configurations. In *Proc. 1 Int. Conference on Supercomputing Systems*, S. 468–475. IEEE Computer Society, 1985.

[Hön90] Hönig, J. Implementierung eines Präprozessors zur Generierung von Watchdogprogrammen. Diplomarbeit im Fach Informatik, Universität Erlangen-Nürnberg, November 1990.

[Hön94] Hönig, J. Zustandssicherung und Wiederanlauf in Multiprozessoren. In Wedekind, H., Hrsg., *Verteilte Systeme*, S. 143–160. BI Wissenschaftsverlag, 1994.

[Hwa93] Hwang, K. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, 1993.

[IEE90] IEEE Std. 1003.1-1990. Standard for Information technology — Portable Operating System Interface (POSIX) — Part 1: System Application Programming Interface (API), 1990.

[Inm88] Inmos Limited. *Transputer Reference Manual*, 1988.

[Int91] Intel Corporation, Supercomputer Systems Division, Beaverton, OR 97006. *Paragon XP/S Product Overview*, 1991.

[IV85] Iyer, R.; Velardi, P. Hardware-Related Software Errors: Measurement and Analysis. *IEEE ToSE*, 11(2), S. 223–231, Februar 1985.

[KGT89] Karlsson, J.; Gunneflo, U.; Torin, J. The Effects of Heavy-ion Induced Single Event Upsets in the MC6809C Microprocessor. In *Fehlertolerierende Rechensysteme*, Band 214, *IFB*, S. 296–307. Springer, 1989.

[KT87] Koo, R.; Toueg, S. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE ToSE*, SE-13(1), S. 23–31, 1987.

[KYA86] Kim, K. H.; You, J. H.; Abouelnaga, A. A scheme for coordinated execution of independently designed recoverable distributed processes. In *Proceedings 16th FTCS*, S. 130–135, 1986.

[Lam78] Lamport, L. Time, Clocks, and the Ordering of events in a Distributed System. *Comm. of the ACM*, 21(7), S. 558–565, Januar 1978.

[Lam88] Lampson, B. W. The stable system. In Lampson, B. W.; Paul, M.; Siegert, H. J., Hrsg., *Distributed Systems: Architecture and Implementation*, S. 254–256. Springer, 1988.

[Lap92] Laprie, J. C., Hrsg. *Dependability: Basic Concepts and Terminology*, Band 5, *Dependable Computing and Fault-Tolerant Systems*. Springer, 1992.

[Leh90] Lehmann, L. *Rekonfiguration und Rückwärtsfehlerbehebung in Multiprozessorsystemen mit begrenzter Nachbarschaft*. Dissertation, Arbeitsberichte des IMMD 23(2), Universität Erlangen-Nürnberg, 1990.

[Li91] Li, K. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proc. of the 1988 Int. Conf. on Parallel Processing*, S. 94–101, 1991.

[LNP90] Li, K.; Naughton, J. F.; Plank, J. S. Real-time, Concurrent Checkpoint for Parallel Programs. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, S. 79–88, März 1990.

[LNP91] Li, K.; Naughton, J. F.; Plank, J. S. Checkpointing Multicomputer Applications. In *Proc. 10th Symposium on Reliable Distributed Systems*, S. 2–12, 1991.

[LSTT94] Linster, C. U.; Stukenbrock, W.; Thiel, Th.; Turowski, S. Das Multiprozessorsystem MEMSY. In Wedekind, H., Hrsg., *Verteilte Systeme*, S. 206–228. BI Wissenschaftsverlag, 1994.

[Lu82] Lu, D. J. Watchdog processors and structural integrity checking. *IEEE Transactions on Computers*, C-31(7), S. 681–685, Juli 1982.

[MH91] Michel, E.; Hohl, W. Concurrent error detection using watchdog processors in the multiprocessor system MEMSY. In *Proceedings of the 5th Int. GI/ITG/GMA Conference on Fault Tolerant Computing Systems*, Band 283, *IFB*, S. 54–64. Springer, 1991.

[Mic92] Michel, E. *Fehlererkennung mit Überwachungsrechnern in Multiprozessorsystemen.* Dissertation, Arbeitsberichte des IMMD 25(6), Universität Erlangen-Nürnberg, 1992.

[MKGT92] Miremadi, G.; Karlsson, J.; Gunneflo, U.; Torin, J. Two software techniques for on-line error detection. In *Proceedings 22nd FTCS*, S. 328–335. IEEE, 1992.

[MM88] Mahmood, A.; McCluskey, E. J. Concurrent error detection using watchdog processors - a survey. *IEEE Transactions on Computers*, 37(2), S. 126–137, Februar 1988.

[Mor86] Moritzen, K. *Softwarewerkzeuge zur Programmierung von Multiprozessorsystemen mit begrenzten Nachbarschaften.* Dissertation, Universität Erlangen-Nürnberg, 1986.

[nCU90] nCUBE Company, Beaverton, OR. *nCUBE 6400 Processor Manual*, 1990.

[Pau88] Pausch, R. *Adding Input and Output to the Transaction Model.* Dissertation, Department of Computer Science, Carnegie Mellon University, 1988.

[PMHH93] Pataricza, A.; Majzik, I.; Hohl, W.; Hönig, J. Watchdog Processors in Parallel Systems. *Microprocessing and Microprogramming*, 39, S. 69–74, 1993.

[PP83] Powell, M. L.; Presotto, D. L. Publishing: A Reliable Broadcast Communication Mechanism. In *Proc. 9th ACM Symp. on OS Principles*, S. 100–109, Oktober 1083.

[Ran75] Randell, B. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, SE-1(2), S. 220–232, 1975.

[Ran79] Randell, B. Software Fault Tolerance. In *EURO IFIP 79*, S. 721–724, 1979.

[Reu81] Reuter, A. *Fehlerbehandlung in Datenbanksystemen.* Carl Hanser Verlag, 1981.

[RT79] Russell, D. L.; Tiedeman, M. J. Multiprocess Recovery Using Conversations. In *Proc. 9th FTCS*, S. 106–109, 1979.

[RW94] Reinwald, B.; Wedekind, H. Transaktionen in verteilten Systemen. In Wedekind, H., Hrsg., *Verteilte Systeme*, S. 121–142. BI Wissenschaftsverlag, 1994.

[Sie90] Siewiorek, D. P. Faults and their manifestation. In B. Simons, A. Spector, Hrsg., *Fault-Tolerant Distributed Computing*, Band 448, *LNCS*, S. 244–261. Springer, 1990.

[Sie93] Sieh, V. Transparente Zustandssicherung und -wiederherstellung im Memsos-Betriebssystem. Diplomarbeit im Fach Informatik, Universität Erlangen-Nürnberg, März 1993.

[Sie94] Sieh, V. Ein Fehlerinjektor auf Basis der UNIX ptrace-Schnittstelle. Interner Bericht des IMMD3, Universität Erlangen-Nürnberg, 1994.

[SMR87] Shrivastava, S.; Mancini, L.; Randell, B. On the Duality of Fault Tolerant System Structures. In Nehmer, J., Hrsg., *Experiences With Distributed Systems*, Band 309, *LNCS*, S. 10–37. Springer, 1987.

[Sos88] Sosnowski, J. Detection of control flow errors using signature and checking instructions. In *Proc. 18th IEEE ITC*, S. 81–88, 1988.

[SPS+94] Sieh, V.; Pataricza, A.; Sallay, B.; Hohl, W.; Hönig, J.; Benyo, B. Fault Injection Based Validation of Fault-tolerant Multiprocessors. In *Proc. of µP'94, 8th Symposium on Computer and Microprocessor Applications*, Oktober 1994. (wird veröffentlicht).

[SS83] Shen, J. P.; Schuette, M. A. On-line self monitoring using signatured instruction streams. In *1983 IEEE International Test Conference*, S. 275–282, 1983.

[SS87] Schuette, M. A.; Shen, J. P. Processor control flow monitoring using signatured instruction streams. *IEEE Transactions on Computers*, 36(3), S. 264–276, März 1987.

[SS91] Schuette, M. A.; Shen, J. P. Exploiting instruction-level resource parallelism for transparent, integrated control-flow monitoring. In *Proceedings 21st FTCS*, S. 318–325, 1991.

[ST82] Sridhar, T.; Thatte, S. M. Concurrent checking of program flow in VLSI processors. In *1982 IEEE International Test Conference*, S. 191–199, 1982.

[STDM82] Schmid, M.; Trapp, R.; Davidoff, A.; Masson, G. Upset Exposure by Means of Abstraction Verification. In *Proc. 12th FTCS*, S. 237–244, 1982.

[SY85] Strom, R. E.; Yemini, S. Optimistic recovery in distributed systems. *ACM ToCS*, S. 204–226, August 1985.

[TA80] Thatte, S. M.; Abraham, J. A. Test Generation for Microprocessors. *IEEE Transactions on Computers*, C-29(6), S. 429–441, Juni 1980.

[Thi91] Thinking Machines Corporation, Cambridge, MA. *The CM-5 Technical Summary*, 1991.

[TS84] Tamir, Y.; Sequin, C. H. Error Recovery in Multiprocessors Using Global Checkpoints. In *Proc. 13th Int. Conf. Parallel Processing*, S. 32–40, 1984.

[WS90a] Wilken, K. D.; Shen, J. P. Concurrent error detection using signature monitoring and encryption. In *1st International Working Conference on Dependable Computers in Critical Applications*. Springer, Wien, 1990.

[WS90b] Wilken, K. D.; Shen, J. P. Continuous signature monitoring: Low-cost concurrent detection of processor control errors. *IEEE Transactions on Computer Aided Design*, 9(6), S. 629–641, Juni 1990.

[YA91] Yuan, S. M.; Agrawala, A. K. Fault-Tolerant Decentralized Commit Protocols. *Journal of Parallel and Distributed Computing*, 13, S. 299–311, 1991.

[YC80] Yau, S. S.; Chen, F. An approach to concurrent control flow checking. *IEEE Transactions on Software Engineering*, SE-6(2), S. 126–137, März 1980.

[You74] Young, J. W. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9), S. 530–531, 1974. Korrektur in 18(2) 1975, S. 95.

Anhang A

A.1 Erwartungswert des Rechenzeitbedarfs in einem System mit Rücksetzen

Es sei zunächst ein System ohne Zustandssicherung angenommen, auf dem ein Auftrag der Länge T_0 zu berechnen ist. Das folgende Verfahren wird zur Fehlerbehebung angenommen: Eine Berechnung mit der Dauer T_0 wird so oft wiederholt, bis sie fehlerfrei terminiert. Fehler werden sofort erkannt, so daß unmittelbar nach dem Erkennen des Fehlers der Wiederholungsbetrieb beginnt. Die Fehler seien nicht korreliert und treten mit der Rate λ auf. Die durchschnittliche zu erwartende Rechenzeit $E[T]$ soll berechnet werden.

Die Wahrscheinlichkeit P_0, das die Berechnung bereits im ersten Versuch korrekt abläuft, errechnet sich zu:

$$P_0 = R(T_0) = e^{-\lambda T_0}$$

Die Wahrscheinlichkeit, daß die Berechnung im ersten Versuch bis zum einem Zeitpunkt t kommt, der Rechner dann ausfällt, nun aber im zweiten Versuch die Berechnung korrekt durchführt berechnet sich zu:

$$P_1(t) = -dR(t) \cdot R(T_0) = e^{-\lambda t}\lambda e^{-\lambda T_0} dt$$

In ähnlicher Weise kann die Wahrscheinlichkeit bestimmt werden, daß die Berechnung erst nach zwei Fehlerversuchen der Dauer t_1 bzw. t_2 terminiert:

$$P_2(t_1, t_2) = \lambda^2 e^{-\lambda t_1} e^{-\lambda t_2} e^{-\lambda T_0} dt_1 dt_2$$

Die zu erwartende durchschnittliche Rechenzeit kann als folgende Summe von Integralen dargestellt werden:

$$\begin{aligned}
E[T] &= T_0 e^{-\lambda T_0} + \\
&+ \lambda e^{-\lambda T_0} \int_0^{T_0} e^{-\lambda t}(T_0 + t)dt + \\
&+ \lambda^2 e^{-\lambda T_0} \int_0^{T_0}\int_0^{T_0} e^{-\lambda t_1} e^{-\lambda t_2}(T_0 + t_1 + t_2)dt_1 dt_2 + \\
&\quad \vdots \\
&+ \lambda^n e^{-\lambda T_0} \int_0^{T_0} \dots \int_0^{T_0} e^{-\lambda t_1} \dots e^{-\lambda t_n}(T_0 + t_1 + \dots + t_n)dt_1 \dots dt_n + \\
&\quad \vdots
\end{aligned}$$

Der jeweils n-te Term $(\lambda^n e^{-\lambda T_0} \dots)$ errechnet sich zu:

$$T_0 e^{-\lambda T_0}(1-e^{-\lambda T_0})^n + n\lambda^n e^{-\lambda T_0}(\frac{e^{-\lambda T_0}}{\lambda^2}(-\lambda T_0 - 1) + \frac{1}{\lambda^2})(\frac{1}{\lambda}(1-e^{-\lambda T_0}))^{n-1}$$

Bei der Summation der Terme ergibt sich folgendes Zwischenergebnis:

$$\begin{aligned} E[T] = \quad & T_0 e^{-\lambda T_0} \sum_{n=0}^{\infty} (1-e^{-\lambda T_0})^n \\ & + e^{-\lambda T_0}(-T_0 e^{-\lambda T_0} - \tfrac{e^{-\lambda T_0}}{\lambda} + \tfrac{1}{\lambda}) \sum_{n=0}^{\infty} n(1-e^{-\lambda T_0})^{n-1} \end{aligned}$$

Beide Reihensummen sind bekannt und konvergieren, die Formel für $E[T]$ vereinfacht sich zu:

$$E[T] = \frac{1}{\lambda}(e^{\lambda T_0} - 1)$$

Wird λ durch λ_l ersetzt, ergibt sich Formel 3.3.

A.2 Manual der MEMSY–Rückwärts-fehlerbehebungs-Bibliothek

NAME

ftlib – MEMSOS rollback recovery package

SYNOPSIS

```
#include <ftlib.h>
int ftinit(id, numtasks)
    unsigned id, numtasks;
void takecp(cpv, cpvcnt)
    struct cpvec *cpv;
    unsigned cpvcnt;
void readcp(cpv, cpvcnt)
    struct cpvec *cpv;
    unsigned cpvcnt;
int ftctl(request, arg)
    int request;
    va_alist arg;
void ftexit();
```

DESCRIPTION

The `ft` library provides rollback recovery subroutines for use with fault-tolerant, parallel applications. The contents of a checkpoint are under full user control. Whenever an error is detected, the application delivers a `SIGHUP` signal to itself. The parent process, usually `appld (8)`, may be configured to restart the application.

`Ftinit()` initialises the library and scans storage for a valid checkpoint of this application. Parameter `numtasks` must be set to the number of processes in this application, `id` must be a number in the range `0 ... numtasks - 1` and uniquely identify the calling process. `Ftinit()` installs handlers for a variety of error–related signals.

`Takecp()` saves checkpoint data, `readcp()` loads checkpoint data. Memory locations to be saved or restored are specified by the `cpv` and `cpvcnt` arguments. `cpv` is a pointer to an array of memory locations of the form

```
struct {
    char *cpv_base;
    int   cpv_len;
} cpvec;
```

`Cpvcnt` must be set to the number of entries in the `cpv` array.

`Ftctl()` may be used to control various parameters of checkpointing and recovery. The `request` code selects a particular function (see below). `Arg` is the argument to this function. The `request` codes and argument types are defined as follows:

`FT_SETFL` Sets miscellaneous flags. `Arg` is any of the following flags ored together:

`FT_CCOW` Create checkpoints with a copy-on-write mechanism. If set, `takecp()` forks a child process each time a checkpoint is saved. A `SIGCHLD`-handler will be installed to capture errors while saving.

`FT_CSYNC` Use synchronous I/O for writing checkpoints. Strongly recommended.

`FT_CWAIT` Wait until the checkpoint has been committed before leaving `takecp()`.

After calling `ftinit`, all flags are unset by default.

`FT_SETTYPE` Sets the type of a checkpoint. `Arg` is an int. After the next checkpoint is written successfully, this value is returned upon restart by `ftinit()`. `Ftinit()` silently replaces a value of 0 with a value of 1.

`FT_SETRP` Installs a restart handler. Instead of delivering `SIGHUP` on errors, the function `arg` will be called. `Arg` is `void (*) ()`.

RETURN VALUES

`Ftinit()` returns 0 if no checkpoint is found, or the `type` of the checkpoint found. All `Ftctl()` requests return -1 in case of errors and set errno accordingly.

FILES

`/usr/tmp/test.*` – checkpoint files

SEE ALSO

`appld(8)`

A.3 Tabellen

Bedingungen		Hauptprozeß			Speicherprozeß		
Test	Puffer	real	usr	sys	real	usr	sys
T/NL	0 MB	32.5	26.1	2.1	68.3	0.0	11.4
T/NL	2 MB	32.7	26.0	1.4	9.1	0.0	1.2
PS/NL	0 MB	32.3	25.9	1.3	34.8	0.3	3.5
PS/NL	2 MB	27.1	25.8	1.2	2.4	0.2	1.2
PS/SY	2 MB	29.2	25.9	1.7	–	–	–

Tabelle A.1: Sicherungszeiten (Poisson, 88100)

	Orig	SF-1	SF-2	SF-5	SF-10	RF-2	RF-5	RF-10
Kontrollflußfehler								
SIGILL	4.6	8.8	7.7	6.6	7.3	8.4	8.0	6.3
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	2.1	1.9	1.9	1.6	1.8	2.8	3.4	2.7
SIGBUS	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGSEGV	78.3	73.0	72.8	76.4	74.1	74.1	74.7	76.2
Signale	85.0	83.9	82.6	84.6	83.3	85.5	86.0	85.6
KF-Fehler	0.0	9.3	9.3	7.3	8.8	7.0	6.2	5.8
kein F.	4.7	3.9	4.1	4.4	4.3	3.6	4.0	4.4
Timeout	0.0	0.0	0.0	0.0	0.2	0.0	0.0	0.0
unerkannt	9.4	2.6	3.6	3.6	3.1	3.5	3.4	3.9
andere F.	0.8	0.3	0.4	0.1	0.2	0.4	0.2	0.3
F. gesamt	10.3	2.9	4.0	3.7	3.5	4.0	3.7	4.2
Registerfehler								
SIGILL	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.9	0.6	0.9	0.6	0.7	0.6	1.4	0.5
SIGBUS	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGSEGV	30.8	27.6	26.9	27.5	29.9	26.1	26.7	24.2
Signale	31.8	28.2	27.9	28.1	30.6	26.6	28.1	24.7
KF-Fehler	0.0	9.5	9.8	10.1	8.8	10.5	8.0	7.6
kein F.	25.7	40.7	41.9	41.4	40.5	32.7	33.1	37.0
Timeout	0.5	0.1	0.1	0.1	0.1	0.1	0.1	0.3
unerkannt	42.0	21.5	20.3	20.3	19.9	30.0	30.6	30.3
andere F.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F. gesamt	42.5	21.6	20.4	20.4	19.9	30.1	30.7	30.6

Tabelle A.2: Fehlerüberdeckung (Poisson, i486)

	Orig	SF-1	SF-2	SF-5	SF-10	RF-2	RF-5	RF-10
Kontrollflußfehler								
SIGILL	10.9	14.5	10.4	9.7	9.0	8.4	14.2	12.6
SIGEMT	1.4	3.6	4.5	2.0	2.9	4.9	2.6	4.0
SIGFPE	0.1	0.3	0.3	0.0	0.3	0.1	0.2	0.0
SIGBUS	17.7	15.3	15.8	16.6	17.2	20.1	16.2	16.5
SIGSEGV	55.6	39.9	44.5	44.6	47.1	42.6	45.3	46.5
Signale	85.7	73.6	75.5	72.9	76.6	76.1	78.5	79.6
KF-Fehler	0.0	17.2	15.2	16.4	12.8	13.5	8.3	6.7
kein F.	4.8	5.1	4.4	4.1	4.0	4.3	5.9	5.5
Timeout	0.5	0.3	0.2	0.5	0.1	0.5	0.0	0.5
unerkannt	8.2	3.8	4.6	5.8	6.3	5.4	6.9	7.6
andere F.	0.8	0.0	0.1	0.3	0.2	0.2	0.4	0.1
F. gesamt	9.5	4.1	4.9	6.6	6.6	6.1	7.3	8.2
Registerfehler								
SIGILL	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGBUS	6.6	3.1	4.6	4.4	3.7	4.4	4.1	5.9
SIGSEGV	19.1	15.4	16.8	15.7	17.0	18.0	16.7	17.1
Signale	25.7	18.5	21.4	20.1	20.7	22.4	20.8	23.0
KF-Fehler	0.0	8.5	7.8	14.1	12.5	10.9	13.2	7.2
kein F.	22.8	31.2	32.2	31.6	30.6	33.2	29.8	32.6
Timeout	3.0	1.7	2.6	3.8	2.8	2.5	2.5	2.4
unerkannt	48.5	40.1	36.0	30.4	33.4	31.0	33.7	34.8
andere F.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F. gesamt	51.5	41.8	38.6	34.2	36.2	33.5	36.2	37.2

Tabelle A.3: Fehlerüberdeckung (Poisson, 68020)

	Orig	SF-1	SF-2	SF-5	SF-10	RF-2	RF-5	RF-10
Kontrollflußfehler								
SIGILL	7.0	7.8	6.5	6.3	8.1	6.3	7.2	6.8
SIGEMT	0.0	0.0	0.1	0.0	0.0	0.2	0.0	0.1
SIGFPE	0.0	0.2	0.0	0.0	0.0	0.0	0.1	0.0
SIGBUS	7.2	4.4	6.2	5.2	4.6	6.7	6.7	7.0
SIGSEGV	56.4	54.9	55.2	55.9	56.7	56.8	55.1	57.3
Signale	70.5	67.5	68.0	67.3	69.3	70.0	69.2	71.2
KF-Fehler	0.0	5.4	6.2	4.5	4.3	3.3	3.8	2.1
kein F.	9.9	13.7	13.4	16.1	14.0	13.6	14.3	14.8
Timeout	0.7	0.9	0.1	0.3	0.1	0.1	0.2	0.6
unerkannt	18.7	12.6	12.3	11.8	12.2	12.8	12.5	11.3
andere F.	0.1	0.0	0.0	0.0	0.0	0.1	0.0	0.1
F. gesamt	19.5	13.4	12.3	12.1	12.3	13.1	12.7	12.0
Registerfehler								
SIGILL	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGBUS	1.0	1.0	0.9	1.0	0.7	1.1	0.6	0.9
SIGSEGV	13.2	17.0	17.2	18.2	17.8	18.5	16.5	16.8
Signale	14.2	18.0	18.1	19.2	18.5	19.7	17.2	17.7
KF-Fehler	0.0	3.2	3.0	3.1	3.7	3.0	1.6	1.6
kein F.	63.0	56.1	55.9	54.9	54.8	54.9	57.2	54.6
Timeout	3.7	2.4	2.5	2.4	2.6	2.4	4.2	3.8
unerkannt	19.0	20.3	20.5	20.5	20.4	20.1	19.9	22.3
andere F.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F. gesamt	22.8	22.7	23.0	22.8	23.0	22.4	24.0	26.1

Tabelle A.4: Fehlerüberdeckung (Poisson, Sparc)

	Orig	SF-1	SF-2	SF-5	SF-10	RF-2	RF-5	RF-10
Kontrollflußfehler								
SIGILL	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.1
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.1	0.0	0.0	0.3	0.1	0.1	0.2	0.1
SIGBUS	30.6	14.4	15.7	13.6	14.4	15.5	16.9	17.1
SIGSEGV	16.3	15.4	15.6	11.2	11.7	15.3	12.7	11.6
Signale	47.6	30.7	31.8	25.7	26.7	31.8	30.8	29.5
KF-Fehler	0.0	26.4	24.2	27.9	26.0	23.0	20.6	18.6
kein F.	14.7	23.6	21.4	22.9	22.8	22.3	18.2	18.5
Timeout	0.9	0.2	0.0	0.3	0.3	0.1	0.2	0.1
unerkannt	36.8	19.2	22.1	23.1	24.1	22.8	29.3	31.3
andere F.	0.0	0.0	0.6	0.1	0.1	0.1	0.9	1.9
F. gesamt	37.7	19.4	22.7	23.5	24.5	22.9	30.4	33.3
Registerfehler								
SIGILL	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGBUS	0.0	0.0	0.0	0.1	0.1	0.0	0.0	0.0
SIGSEGV	5.0	2.7	4.6	4.4	5.4	4.3	4.1	3.6
Signale	5.0	2.7	4.6	4.5	5.4	4.3	4.1	3.6
KF-Fehler	0.0	4.2	3.4	5.0	4.5	3.1	2.6	3.0
kein F.	44.0	62.7	58.5	58.7	57.8	62.2	62.3	63.2
Timeout	0.9	1.6	1.5	1.1	1.3	0.1	0.0	0.0
unerkannt	50.0	28.9	32.0	30.7	31.0	30.3	30.9	30.2
andere F.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F. gesamt	51.0	30.5	33.5	31.8	32.3	30.4	31.0	30.2

Tabelle A.5: Fehlerüberdeckung (Poisson, 88100)

	Orig	SF-1	SF-2	SF-5	SF-10	RF-2	RF-5	RF-10
Kontrollflußfehler								
SIGILL	6.5	7.7	6.8	9.5	8.9	5.8	11.2	7.3
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	5.2	2.5	2.4	2.5	2.5	2.6	2.9	3.3
SIGBUS	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGSEGV	82.2	75.8	76.8	76.2	74.8	78.7	75.2	78.7
Signale	94.0	86.1	86.0	88.2	86.2	87.1	89.2	89.3
KF-Fehler	0.0	8.4	7.8	5.7	7.2	6.5	4.6	3.7
kein F.	1.4	1.9	1.8	1.9	2.0	2.7	2.5	2.5
Timeout	0.1	0.0	0.1	0.0	0.0	0.1	0.0	0.0
unerkannt	3.9	3.0	4.0	3.5	4.3	3.1	3.1	4.1
andere F.	0.7	0.6	0.4	0.8	0.1	0.4	0.6	0.5
F. gesamt	4.6	3.5	4.4	4.3	4.5	3.7	3.6	4.5
Registerfehler								
SIGILL	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	5.2	4.2	4.7	3.2	6.0	5.2	2.9	6.5
SIGBUS	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGSEGV	38.2	35.5	37.5	34.6	37.1	34.6	37.2	39.9
Signale	43.5	39.7	42.1	37.9	43.1	39.8	40.1	46.4
KF-Fehler	0.0	4.0	2.0	0.8	0.8	2.6	0.9	0.4
kein F.	44.1	44.6	45.7	48.8	43.2	45.5	47.5	41.2
Timeout	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
unerkannt	12.4	11.7	10.1	12.5	12.8	12.2	11.4	12.0
andere F.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F. gesamt	12.4	11.7	10.1	12.5	12.8	12.2	11.4	12.0

Tabelle A.6: Fehlerüberdeckung (Navier-Stokes, i486)

	Orig	SF-1	SF-2	SF-5	SF-10	RF-2	RF-5	RF-10
Kontrollflußfehler								
SIGILL	14.2	11.8	12.5	12.3	9.9	9.2	9.2	10.4
SIGEMT	5.2	3.2	2.9	4.5	3.9	2.8	4.6	4.5
SIGFPE	0.2	0.1	0.1	0.1	0.1	0.1	0.0	0.1
SIGBUS	17.9	20.1	23.4	20.1	21.1	22.3	21.8	21.6
SIGSEGV	45.2	42.5	39.0	42.4	41.9	43.6	43.9	40.2
Signale	82.8	77.8	77.8	79.3	76.8	78.0	79.5	76.8
KF-Fehler	0.0	7.3	6.8	7.2	7.5	7.6	6.0	7.0
kein F.	4.2	5.4	4.8	3.3	5.3	5.0	4.7	5.5
Timeout	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
unerkannt	12.8	9.3	10.4	10.1	10.3	8.8	9.1	10.7
andere F.	0.1	0.0	0.0	0.0	0.0	0.6	0.8	0.0
F. gesamt	13.0	9.4	10.5	10.1	10.3	9.4	9.9	10.7
Registerfehler								
SIGILL	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0
SIGBUS	9.7	9.3	8.5	8.7	11.5	8.6	8.4	8.5
SIGSEGV	19.9	18.3	21.6	21.2	20.9	18.3	25.0	20.6
Signale	29.6	27.7	30.1	30.0	32.4	26.9	33.4	29.1
KF-Fehler	0.0	1.4	1.9	2.9	2.7	1.6	3.2	2.2
kein F.	57.0	53.6	52.0	53.4	52.5	54.3	49.1	53.3
Timeout	0.0	0.2	0.1	0.0	0.0	0.1	0.0	0.0
unerkannt	13.4	17.1	15.9	13.7	12.4	17.1	14.3	15.4
andere F.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F. gesamt	13.4	17.3	16.0	13.7	12.4	17.2	14.3	15.4

Tabelle A.7: Fehlerüberdeckung (Navier-Stokes, 68020)

	Orig	SF-1	SF-2	SF-5	SF-10	RF-2	RF-5	RF-10
Kontrollflußfehler								
SIGILL	7.2	13.6	8.0	6.7	8.7	9.1	8.4	7.0
SIGEMT	0.0	0.1	0.0	0.1	0.0	0.1	0.0	0.1
SIGFPE	0.0	0.1	0.1	0.1	0.0	0.0	0.0	0.1
SIGBUS	5.5	4.0	3.6	6.2	5.2	4.8	3.9	4.3
SIGSEGV	51.5	43.4	53.8	54.6	50.0	51.6	54.0	54.9
Signale	64.2	61.2	65.4	67.6	63.9	65.5	66.2	66.3
KF-Fehler	0.0	3.7	2.7	2.5	2.9	2.8	2.3	1.2
kein F.	17.1	19.8	15.1	14.4	16.2	15.2	16.6	16.8
Timeout	0.0	0.1	0.0	0.0	0.1	0.0	0.0	0.0
unerkannt	18.7	15.3	16.8	15.5	16.9	16.4	14.9	15.6
andere F.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F. gesamt	18.7	15.3	16.8	15.5	17.0	16.4	14.9	15.6
Registerfehler								
SIGILL	0.3	0.2	0.1	0.2	0.0	0.0	0.0	0.1
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGBUS	5.9	5.1	5.6	4.7	5.8	5.3	5.2	5.3
SIGSEGV	20.8	22.7	24.6	20.7	20.4	21.9	20.7	20.1
Signale	27.0	28.0	30.3	25.6	26.2	27.3	25.9	25.5
KF-Fehler	0.0	2.3	0.9	1.3	1.0	1.0	1.3	0.8
kein F.	62.8	62.1	60.6	66.2	64.3	62.9	63.6	65.8
Timeout	0.3	0.1	0.2	0.2	0.0	0.6	0.1	0.3
unerkannt	9.9	7.5	8.0	6.7	8.5	8.2	9.1	7.6
andere F.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F. gesamt	10.2	7.6	8.2	6.9	8.5	8.8	9.2	7.9

Tabelle A.8: Fehlerüberdeckung (Navier-Stokes, Sparc)

	Orig	SF-1	SF-2	SF-5	SF-10	RF-2	RF-5	RF-10
Kontrollflußfehler								
SIGILL	0.2	0.0	0.1	0.0	0.1	0.1	0.0	0.0
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.4	0.0	0.0	0.0	0.1	0.1	0.0	0.1
SIGBUS	37.7	24.0	25.8	26.5	27.1	25.2	27.0	28.9
SIGSEGV	18.3	18.5	19.9	20.4	19.9	21.4	21.9	22.9
Signale	57.2	42.7	46.0	47.0	47.4	47.0	49.1	52.1
KF-Fehler	0.0	16.9	12.4	10.6	10.2	11.6	8.9	8.0
kein F.	10.1	13.1	12.4	11.9	11.4	11.6	12.5	11.5
Timeout	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0
unerkannt	32.5	27.3	29.2	30.4	31.1	29.8	29.4	28.3
andere F.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F. gesamt	32.7	27.3	29.3	30.4	31.1	29.8	29.5	28.3
Registerfehler								
SIGILL	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGBUS	3.6	3.9	4.0	2.9	3.4	4.1	3.1	2.8
SIGSEGV	3.9	4.6	3.8	2.7	3.3	4.4	3.5	3.5
Signale	7.4	8.5	7.8	5.6	6.7	8.5	6.7	6.3
KF-Fehler	0.0	2.8	2.5	1.7	2.2	2.5	1.1	1.1
kein F.	71.8	70.0	73.2	74.5	74.1	70.6	75.4	73.8
Timeout	0.2	0.1	0.1	0.0	0.5	0.2	0.3	0.2
unerkannt	20.6	18.6	16.4	18.2	16.6	18.2	16.5	18.6
andere F.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F. gesamt	20.8	18.6	16.6	18.2	17.1	18.5	16.8	18.8

Tabelle A.9: Fehlerüberdeckung (Navier-Stokes, 88100)

	Orig	SF-1	SF-2	SF-5	SF-10	RF-2	RF-5	RF-10
Kontrollflußfehler								
SIGILL	3.5	5.5	4.3	4.1	3.8	3.8	2.9	3.2
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.3	0.2	0.5	0.1	0.1	0.1	0.1	0.0
SIGBUS	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGSEGV	81.0	77.2	79.3	76.0	77.8	81.2	78.1	79.2
Signale	84.9	82.8	84.2	80.2	81.7	85.0	81.3	82.4
KF-Fehler	0.0	7.3	6.2	6.8	6.2	5.6	3.3	3.3
kein F.	7.2	6.1	5.5	9.1	8.2	6.1	10.6	8.3
Timeout	0.3	0.0	0.1	0.0	0.1	0.0	0.3	0.2
unerkannt	4.0	1.7	1.9	1.6	2.2	1.4	2.5	2.9
andere F.	3.5	2.0	2.2	2.3	1.6	1.9	1.9	2.9
F. gesamt	7.8	3.8	4.2	3.9	3.9	3.3	4.8	6.0
Registerfehler								
SIGILL	0.1	0.2	0.6	0.6	0.2	0.5	0.2	0.5
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.5	0.6	0.2	0.2	0.3	0.4	0.3	0.6
SIGBUS	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGSEGV	18.3	25.7	27.3	25.6	26.2	31.7	20.7	23.7
Signale	18.9	26.5	28.1	26.4	26.7	32.6	21.2	24.8
KF-Fehler	0.0	7.2	8.2	6.7	5.5	5.7	7.4	5.6
kein F.	63.2	55.2	52.1	54.1	54.7	50.3	59.8	55.6
Timeout	0.1	0.3	0.1	0.0	0.0	0.1	0.0	0.1
unerkannt	17.7	10.7	11.5	12.8	12.9	11.3	11.6	13.9
andere F.	0.1	0.1	0.0	0.0	0.2	0.0	0.0	0.0
F. gesamt	17.9	11.1	11.6	12.8	13.1	11.4	11.6	14.0

Tabelle A.10: Fehlerüberdeckung (Dhrystone, i486)

	Orig	SF-1	SF-2	SF-5	SF-10	RF-2	RF-5	RF-10
Kontrollflußfehler								
SIGILL	6.2	4.0	3.9	3.9	3.5	4.0	4.4	3.7
SIGEMT	3.3	2.9	3.7	3.4	3.5	3.2	3.3	4.5
SIGFPE	0.0	0.1	0.0	0.0	0.1	0.0	0.2	0.2
SIGBUS	15.5	2.9	15.8	15.3	15.6	14.7	15.8	14.0
SIGSEGV	53.2	59.0	52.8	53.1	54.3	53.2	51.7	51.3
Signale	78.2	69.0	76.2	75.9	77.0	75.2	75.4	73.7
KF-Fehler	0.0	8.1	7.4	9.0	8.0	9.6	9.0	9.8
kein F.	10.2	19.4	11.1	10.1	10.0	10.3	10.4	11.7
Timeout	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
unerkannt	10.4	3.2	4.9	4.4	4.7	4.2	4.8	4.5
andere F.	1.2	0.3	0.5	0.7	0.3	0.7	0.3	0.4
F. gesamt	11.6	3.5	5.4	5.1	5.0	4.9	5.2	4.8
Registerfehler								
SIGILL	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGBUS	1.8	2.0	1.9	1.5	1.4	2.0	1.4	1.4
SIGSEGV	4.4	4.6	4.8	5.0	4.8	5.0	4.4	4.7
Signale	6.2	6.6	6.8	6.5	6.2	7.0	5.8	6.1
KF-Fehler	0.0	2.7	2.1	1.8	2.0	2.1	1.7	1.7
kein F.	76.0	73.3	74.8	73.9	75.0	74.5	75.2	74.5
Timeout	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
unerkannt	6.2	7.2	7.1	7.6	6.9	7.0	7.2	7.3
andere F.	11.6	10.3	9.3	10.2	9.9	9.4	10.1	10.3
F. gesamt	17.8	17.4	16.4	17.8	16.8	16.4	17.3	17.6

Tabelle A.11: Fehlerüberdeckung (Dhrystone, 68020)

	Orig	SF-1	SF-2	SF-5	SF-10	RF-2	RF-5	RF-10
Kontrollflußfehler								
SIGILL	2.6	4.7	3.8	3.8	2.6	2.9	3.8	3.1
SIGEMT	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.3	0.2	0.0	0.2	0.0	0.2	0.0	0.2
SIGBUS	3.3	2.2	3.2	2.6	3.1	2.7	2.8	2.0
SIGSEGV	65.2	55.2	57.1	52.4	54.9	57.3	66.3	66.6
Signale	71.4	62.5	64.1	59.0	60.6	63.1	72.9	71.9
KF-Fehler	0.0	7.5	5.2	6.0	6.5	6.1	3.6	2.5
kein F.	17.2	21.4	20.3	24.5	24.0	21.7	14.6	15.6
Timeout	0.3	0.0	0.1	0.0	0.0	0.1	0.2	0.2
unerkannt	10.8	8.0	9.4	10.2	8.3	8.8	8.2	8.9
andere F.	0.3	0.6	0.9	0.3	0.6	0.2	0.5	0.9
F. gesamt	11.4	8.6	10.4	10.5	8.9	9.1	8.9	10.0
Registerfehler								
SIGILL	0.1	0.0	0.0	0.0	0.1	0.0	0.0	0.0
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGBUS	0.8	0.4	0.4	0.5	0.2	0.4	0.5	0.5
SIGSEGV	8.1	7.1	7.1	6.3	5.5	7.0	5.4	5.4
Signale	9.0	7.6	7.5	6.8	5.8	7.5	5.9	5.9
KF-Fehler	0.0	2.3	2.5	1.9	1.9	2.0	1.5	1.4
kein F.	82.9	84.7	85.6	85.5	87.8	85.2	87.9	87.8
Timeout	1.1	0.8	1.0	0.9	1.1	1.0	0.9	0.8
unerkannt	7.0	4.6	3.4	4.8	3.5	4.3	3.8	4.0
andere F.	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0
F. gesamt	8.1	5.4	4.4	5.8	4.5	5.3	4.7	4.8

Tabelle A.12: Fehlerüberdeckung (Dhrystone, Sparc)

	Orig	SF-1	SF-2	SF-5	SF-10	RF-2	RF-5	RF-10
Kontrollflußfehler								
SIGILL	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.1
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.8	0.5	0.8	0.8	0.8	0.8	0.9	0.7
SIGBUS	24.3	17.2	19.3	20.3	21.6	20.5	20.9	19.3
SIGSEGV	13.6	13.5	14.0	16.2	15.9	16.4	19.8	19.1
Signale	39.1	32.2	35.1	38.2	39.2	38.7	42.3	40.2
KF-Fehler	0.0	25.0	21.2	17.1	17.7	18.9	13.9	12.8
kein F.	34.5	28.1	28.9	28.3	28.2	27.1	27.3	29.3
Timeout	1.2	0.7	0.6	0.7	0.8	0.8	0.9	0.7
unerkannt	23.6	13.0	13.1	14.9	13.2	13.7	14.4	16.2
andere F.	1.6	1.0	1.1	0.8	0.9	0.8	1.1	0.8
F. gesamt	26.4	14.6	14.8	16.4	14.9	15.3	16.4	17.7
Registerfehler								
SIGILL	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.0	0.0	0.0	0.0	0.0	0.1	0.0	0.0
SIGBUS	0.9	0.5	0.3	0.5	0.4	0.6	0.6	0.6
SIGSEGV	1.3	1.3	1.0	1.2	1.2	1.0	1.5	0.9
Signale	2.2	1.8	1.4	1.7	1.6	1.7	2.1	1.5
KF-Fehler	0.0	5.4	5.1	6.0	6.2	6.3	5.5	3.6
kein F.	79.2	85.3	85.1	84.4	84.3	84.8	83.5	79.6
Timeout	0.1	0.1	0.0	0.0	0.0	0.0	0.0	0.1
unerkannt	18.3	7.4	8.3	7.8	7.8	7.2	8.8	15.1
andere F.	0.2	0.0	0.2	0.1	0.1	0.1	0.1	0.1
F. gesamt	18.6	7.5	8.4	7.8	7.9	7.2	8.9	15.3

Tabelle A.13: Fehlerüberdeckung (Dhrystone, 88100)

	Orig	SF-1	SF-2	SF-5	SF-10	RF-2	RF-5	RF-10
Kontrollflußfehler								
SIGILL	3.4	5.9	5.2	4.1	3.8	3.7	4.0	3.2
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	4.2	0.7	1.9	2.1	1.4	1.9	2.5	3.4
SIGBUS	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGSEGV	79.8	84.9	81.9	83.8	83.8	83.9	83.2	81.2
Signale	87.3	91.5	88.9	90.0	89.0	89.5	89.8	87.8
KF-Fehler	0.0	5.6	6.8	5.7	5.5	6.8	3.9	4.8
kein F.	10.4	1.8	3.5	3.6	4.2	3.0	5.2	6.0
Timeout	0.0	0.1	0.0	0.0	0.1	0.1	0.1	0.2
unerkannt	0.8	0.5	0.1	0.2	0.6	0.2	0.7	0.1
andere F.	1.4	0.5	0.7	0.4	0.5	0.3	0.5	0.8
F. gesamt	2.2	1.1	0.8	0.7	1.2	0.7	1.2	1.2
Registerfehler								
SIGILL	0.2	0.3	0.5	0.2	0.2	0.3	0.1	0.4
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.2	0.0	0.1	0.1	0.2	0.1	0.1	0.2
SIGBUS	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGSEGV	18.4	17.8	19.8	20.2	19.5	21.1	23.4	20.6
Signale	18.9	18.1	20.2	20.6	20.0	21.6	23.7	21.2
KF-Fehler	0.0	6.1	7.7	6.7	6.5	7.0	4.8	6.0
kein F.	57.1	50.4	50.4	46.3	48.2	48.0	45.8	48.1
Timeout	10.7	11.2	10.3	12.2	11.1	8.0	11.2	10.8
unerkannt	13.4	14.2	11.4	14.3	14.2	15.3	14.6	13.8
andere F.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F. gesamt	24.1	25.4	21.8	26.4	25.3	23.4	25.8	24.6

Tabelle A.14: Fehlerüberdeckung (Whetstone, i486)

	Orig	SF-1	SF-2	SF-5	SF-10	RF-2	RF-5	RF-10
Kontrollflußfehler								
SIGILL	6.2	6.8	4.9	4.3	6.9	4.9	5.5	4.4
SIGEMT	3.7	2.9	2.9	3.5	2.6	2.5	5.0	3.2
SIGFPE	0.0	1.7	0.0	0.1	0.1	0.1	0.0	0.1
SIGBUS	19.6	19.4	19.3	19.4	16.5	19.1	17.7	19.8
SIGSEGV	50.1	49.9	49.4	53.3	52.3	51.9	53.1	56.8
Signale	79.6	80.7	76.5	80.6	78.4	78.5	81.3	84.3
KF-Fehler	0.0	8.7	9.9	8.4	9.3	8.3	7.0	5.8
kein F.	14.3	6.9	10.1	8.1	9.2	9.9	7.7	7.5
Timeout	0.1	0.3	0.2	0.0	0.3	0.1	0.0	0.1
unerkannt	3.9	2.1	2.0	2.1	1.4	2.3	2.2	1.3
andere F.	2.1	1.3	1.3	0.8	1.4	0.9	1.8	1.0
F. gesamt	6.1	3.7	3.5	2.9	3.1	3.3	4.0	2.4
Registerfehler								
SIGILL	0.2	0.2	0.0	0.0	0.4	0.5	0.1	0.1
SIGEMT	0.6	0.2	0.1	0.0	0.3	0.2	0.1	0.2
SIGFPE	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0
SIGBUS	1.3	0.8	1.6	1.6	2.0	1.4	1.8	1.5
SIGSEGV	5.1	3.2	3.5	4.0	3.8	4.3	5.1	4.1
Signale	7.2	4.4	5.2	5.7	6.5	6.4	7.1	5.9
KF-Fehler	0.0	1.6	1.9	3.5	2.0	2.7	3.0	1.6
kein F.	83.2	83.6	82.5	80.7	84.4	80.1	79.5	85.9
Timeout	3.7	4.1	4.7	3.6	2.6	3.4	3.6	1.4
unerkannt	4.5	5.7	4.8	5.6	3.7	7.0	6.5	4.5
andere F.	1.4	0.6	0.9	0.9	0.8	0.4	0.3	0.7
F. gesamt	9.6	10.4	10.4	10.1	7.1	10.8	10.4	6.6

Tabelle A.15: Fehlerüberdeckung (Whetstone, 68020)

	Orig	SF-1	SF-2	SF-5	SF-10	RF-2	RF-5	RF-10
Kontrollflußfehler								
SIGILL	5.2	4.2	4.9	6.7	6.8	5.9	7.0	7.0
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGBUS	2.6	1.9	2.4	2.5	2.9	2.4	2.9	2.6
SIGSEGV	47.2	57.9	58.3	59.2	59.8	58.9	59.5	60.4
Signale	55.0	64.0	65.8	68.5	69.5	67.2	69.2	70.1
KF-Fehler	0.0	10.4	9.2	6.8	6.1	7.9	5.3	5.1
kein F.	41.9	23.6	23.0	21.8	22.5	22.9	23.6	22.7
Timeout	0.3	0.1	0.1	0.1	0.0	0.2	0.0	0.1
unerkannt	1.6	1.0	1.0	1.2	0.8	0.6	1.1	1.1
andere F.	1.2	0.9	0.9	1.6	1.1	1.1	0.7	0.8
F. gesamt	3.1	2.0	2.0	2.9	1.9	2.0	1.8	2.1
Registerfehler								
SIGILL	0.1	0.0	0.1	0.0	0.0	0.0	0.1	0.1
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGBUS	2.2	0.9	1.1	0.8	1.2	1.2	1.2	0.9
SIGSEGV	19.1	11.4	9.9	10.9	10.4	9.2	9.7	9.8
Signale	21.5	12.3	11.0	11.7	11.7	10.4	10.9	10.8
KF-Fehler	0.0	3.0	2.4	3.1	1.4	3.0	2.9	1.4
kein F.	72.0	83.2	84.0	82.0	83.9	83.5	83.5	84.8
Timeout	3.1	0.2	0.5	1.4	1.4	1.9	1.1	1.6
unerkannt	3.2	1.2	2.2	1.8	1.8	1.1	1.6	1.2
andere F.	0.2	0.0	0.0	0.0	0.0	0.1	0.0	0.0
F. gesamt	6.5	1.4	2.7	3.2	3.1	3.1	2.8	2.9

Tabelle A.16: Fehlerüberdeckung (Whetstone, Sparc)

	Orig	SF-1	SF-2	SF-5	SF-10	RF-2	RF-5	RF-10
Kontrollflußfehler								
SIGILL	0.1	0.0	0.4	0.1	0.1	0.3	0.2	0.1
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.2	0.1	1.2	0.3	0.2	0.1	0.0	0.1
SIGBUS	29.1	23.1	24.9	24.7	25.7	26.0	28.8	29.8
SIGSEGV	18.9	16.7	19.4	17.7	17.5	21.1	14.6	13.1
Signale	48.8	40.0	46.1	42.9	43.7	47.6	43.9	43.6
KF-Fehler	0.0	23.5	18.8	21.1	18.0	17.1	18.5	17.2
kein F.	40.1	31.1	30.2	29.5	31.7	30.4	31.3	32.5
Timeout	2.4	0.2	0.3	0.5	0.5	0.5	0.4	0.3
unerkannt	8.3	4.8	4.7	5.6	6.0	4.5	5.8	6.3
andere F.	0.4	0.4	0.0	0.3	0.1	0.1	0.1	0.0
F. gesamt	11.1	5.4	5.0	6.4	6.6	5.0	6.3	6.6
Registerfehler								
SIGILL	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGEMT	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGFPE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SIGBUS	0.5	0.2	0.2	0.2	0.2	0.3	0.1	0.3
SIGSEGV	1.3	0.5	0.9	1.4	1.3	1.6	1.3	1.2
Signale	1.8	0.7	1.1	1.6	1.6	2.0	1.4	1.5
KF-Fehler	0.0	4.7	5.0	4.1	3.8	5.6	4.5	4.3
kein F.	88.1	88.6	88.8	87.6	87.8	88.0	87.6	87.0
Timeout	1.9	0.9	0.9	1.0	1.1	0.5	1.1	1.3
unerkannt	8.0	5.1	4.2	5.6	5.8	3.8	5.4	5.9
andere F.	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0
F. gesamt	10.0	6.0	5.1	6.7	6.8	4.4	6.5	7.2

Tabelle A.17: Fehlerüberdeckung (Whetstone, 88100)

Parameter	Laufzeitmehraufwand				Speichermehraufwand			
	i486	68020	Sparc	88100	i486	68020	Sparc	88100
SF-1	9.0%	20.1%	41.2%	31.0%	276%	283%	264%	225%
SF-2	13.5%	19.0%	29.0%	24.1%	156%	161%	158%	135%
SF-5	3.7%	16.7%	24.9%	23.4%	117%	120%	122%	104%
SF-10	11.1%	17.0%	25.1%	23.6%	106%	109%	113%	96%
RF-2	6.1%	15.5%	24.9%	17.3%	122%	125%	125%	108%
RF-5	2.1%	12.4%	14.6%	15.1%	69%	68%	70%	63%
RF-10	1.4%	12.2%	9.6%	10.4%	55%	53%	59%	53%
Orig (100%)	63.0s	156.1s	22.0s	19.7s	3.8k	3.4k	4.4k	4.7k

Tabelle A.18: Mehraufwand (Poisson)

Parameter	Laufzeitmehraufwand				Speichermehraufwand			
	i486	68020	Sparc	88100	i486	68020	Sparc	88100
SF-1	15.0%	9.3%	6.2%	13.5%	484%	494%	347%	425%
SF-2	11.2%	6.9%	6.1%	9.4%	301%	304%	219%	267%
SF-5	7.1%	6.3%	4.1%	7.9%	226%	228%	166%	204%
SF-10	2.9%	4.4%	3.0%	6.0%	205%	209%	153%	186%
RF-2	9.2%	6.8%	4.1%	9.5%	211%	210%	155%	187%
RF-5	5.8%	5.0%	4.9%	9.0%	96%	94%	72%	89%
RF-10	4.8%	3.6%	5.3%	5.4%	68%	67%	51%	64%
Orig (100%)	28.7s	161.5s	23.4s	14.4s	33.0k	30.2k	47.4k	36.0k

Tabelle A.19: Mehraufwand (Navier-Stokes)

Parameter	Laufzeitmehraufwand				Speichermehraufwand			
	i486	68020	Sparc	88100	i486	68020	Sparc	88100
SF-1	54.1%	47.3%	37.4%	83.4%	376%	358%	312%	260%
SF-2	40.8%	39.3%	31.1%	69.6%	225%	216%	193%	165%
SF-5	30.7%	33.1%	63.6%	58.7%	170%	159%	143%	125%
SF-10	29.5%	32.8%	62.6%	57.7%	155%	147%	134%	117%
RF-2	34.8%	37.3%	61.4%	64.8%	174%	169%	152%	131%
RF-5	22.3%	24.9%	57.4%	43.8%	82%	78%	74%	68%
RF-10	17.7%	22.4%	48.8%	39.1%	57%	55%	52%	51%
Orig (100%)	8.7s	44.9s	4.4s	4.0s	2.0k	2.0k	2.7k	3.0k

Tabelle A.20: Mehraufwand (Dhrystone)

Parameter	Laufzeitmehraufwand				Speichermehraufwand			
	i486	68020	Sparc	88100	i486	68020	Sparc	88100
SF-1	64.2%	61.5%	54.6%	104.1%	595%	558%	583%	560%
SF-2	50.2%	49.0%	31.4%	82.2%	287%	268%	287%	274%
SF-5	31.4%	38.5%	14.2%	59.0%	161%	151%	163%	156%
SF-10	32.9%	39.3%	14.2%	58.5%	136%	127%	141%	133%
RF-2	40.9%	47.9%	25.5%	75.1%	209%	196%	212%	202%
RF-5	22.8%	34.5%	13.5%	51.6%	76%	72%	79%	76%
RF-10	15.0%	27.2%	10.7%	35.8%	37%	37%	42%	41%
Orig (100%)	3.6s	20.7s	1.7s	2.0s	3.0k	3.1k	3.3k	3.2k

Tabelle A.21: Mehraufwand (Whetstone)