Dissertation
zur Erlangung des akademischen Grades
Dr. rer. nat.

Virtualisierung von Mehrprozessorsystemen mit Echtzeitanwendungen

vorgelegt von Robert Kaiser

am 11. Febuar 2009

Datum der wissenschaftlichen Aussprache: 26. Mai 2009

Referent: Prof. Dr. Dieter Zöbel, Universität Koblenz-Landau

Korreferenten: Prof. Dr. Reinhold Kröger, Fachhochschule Wiesbaden

Prof. Dr. Wolfgang Schröder-Preikschat, Universität Erlangen-Nürnberg

Abstract

Due to advances in microchip technology, embedded systems today provide computational resources that, only a few years ago, would have taken a server- or workstation-class machine to provide. Also, since the introduction of ”multicore” processors, embedded multiprocessing is now becoming a viable perspective, thus promising another boost to embedded system performance. In order to put this potential to good use, functions that were previously distributed across multiple separate computing systems, must now be integrated into a smaller set of more powerful computers. This, however, leads to increased per-system software complexity. Programs that previously resided in separate machines now coexist in a common machine. Without special precautions, this leads to numerous possibilities for undesired interactions between the programs which, as a whole, are almost impossible to keep under control.

In the server field, quite similar problems have been solved by virtualisation: By dividing a physical machine into multiple virtual machines, programs can be effectively isolated from each other, so faults can not propagate throughout the system. Moreover, multiprocessor architectures have been commonly used for parallel computation in the server field, often in combination with virtualisation.

Thus, applying these well-known techniques to embedded systems now would seem like an obvious choice. However simply re-applying them ”as-is” does not work well. Both the concept of virtualisation as well as the concept of parallel computing need to be reconsidered as a third concept is added to the picture: Embedded systems commonly need to support real-time computation.

This work is about three subjects: Virtualisation, real-time computing and parallel computing. Taken by itself, each of these subjects has already been well-researched, however, when considering all three together, as is necessary when looking at embedded systems, numerous questions as well as new possibilities arise.

In this work we develop models describing the behaviour and requirements of real-time applications which execute in a hierarchy of processes as they do when

running in a virtual machine. Also, the real-time capabilities of existing virtual machines are evaluated and new interfaces for virtualisation of multiprocessor machines which take into account the characteristics of embedded systems –specifically real-time computing– are defined, implemented and tested. This enables safe, secure and efficient coexistence of programs with largely differing time constraints within separate virtual machines on a single, common multiprocessor computer.

Zusammenfassung

Aufgrund der Fortschritte in der Chiptechnologie verfügen heutige eingebettete Systeme über Rechenleistungen, wie sie noch vor wenigen Jahren nur bei Rechnern der Server- oder Workstation-Klasse anzutreffen waren. Durch die Einführung der „Multicore"-Prozessoren werden nun auch eingebettete Mehrprozessorsysteme realistisch, was einen weiteren Leistungsschub erwarten lässt. Um das Potenzial dieser Systeme zu nutzen, müssen Funktionen, die früher auf vielen getrennten Rechnern angesiedelt waren, nun durch einige wenige, dafür leistungsfähigere Rechensysteme erbracht werden. Dabei steigt die Komplexität der Software jedes einzelnen Systems an. Programme, die früher auf verschiedenen Rechnern gearbeitet haben, müssen nun in einem gemeinsamen Rechner koexistieren. Ohne geeignete Schutzmaßnahmen ergeben sich dadurch vielfältige Möglichkeiten unerwünschter Wechselwirkungen, die in ihrer Gesamtheit kaum noch beherrschbar sind.

Im Bereich der Server hat sich bei ähnlicher Problemlage die Virtualisierung als geeignetes Mittel erwiesen: Durch die Aufteilung einer physischen Maschine in mehrere virtuelle Maschinen können Barrieren zwischen Programmen aufgebaut werden, die eine unkontrollierte Fehlerausbreitung unterbinden. Zudem gibt es im Serverbereich schon seit langem etablierte Vorgehensweisen zur Parallelverarbeitung auf Mehrprozessorrechnern, auch und gerade in Verbindung mit Virtualisierung.

Es liegt nahe, diese erprobten Konzepte nun auch auf eingebettete Systeme anzuwenden, doch eine einfache Übernahme der Techniken erweist sich als nicht tragfähig. Der Grund hierfür liegt in der zusätzlichen Forderung nach Echtzeitverarbeitung, die bei eingebettetem Systemen in der Regel erhoben wird.

Diese Arbeit bewegt sich im Spannungsfeld dreier Gebiete: Virtualisierung, Echtzeitverarbeitung und Parallelverarbeitung. Jedes dieser Gebiete gilt für sich genommen als weitgehend erforscht, doch ergeben sich bei ihrer gemeinsamen Betrachtung zahlreiche neue Fragestellungen und Möglichkeiten.

In dieser Arbeit werden dazu Modelle zur Beschreibung von Echtzeitanwendungen innerhalb der Prozesshierarchie einer Virtualisierungsumgebung entwickelt.

Bestehende Schnittstellen zur Virtualisierung werden auf ihre Möglichkeiten zur Echtzeitverarbeitung untersucht, und es werden neue Schnittstellen zur Virtualisierung auf Mehrprozessormaschinen geschaffen und erprobt, die die spezifischen Anforderungen eingebetteter Systeme –insbesondere die Echtzeitfähigkeit– berücksichtigen. Damit wird eine sichere und effiziente Koexistenz von Programmen mit unterschiedlich harten Zeitanforderungen in getrennten virtuellen Maschinen auf einem gemeinsamen Mehrprozessorrechner ermöglicht.

Danksagung

An dieser Stelle möchte ich mich bei einigen Personen bedanken, ohne deren Hilfe und Unterstützung diese Arbeit wohl kaum zustande gekommen wäre.

Zunächst danke ich Prof. Dr. Dieter Zöbel dafür, dass er die Betreuung meiner Promotion übernommen hat. Es ist mir nicht immer leicht gefallen, seinen Ratschlägen zu folgen, aber im Nachhinein kann ich nur feststellen, dass er mich damit immer wieder zu neuen Ideen angespornt und so erheblich zur Verbesserung dieser Arbeit beigetragen hat.

Prof. Dr. Wolfgang Schröder-Preikschat danke ich, dass er so kurzfristig und unproblematisch bereit war, als Berichterstatter die Begutachtung meiner Arbeit zu übernehmen.

Mein besonderer Dank gilt Prof. Dr. Reinhold Kröger, der mir überhaupt erst die Möglichkeit aufgezeigt hat, in meinem Alter noch zu promovieren, und der mich bei diesem Vorhaben mit viel persönlichem Engagement unterstützt hat. Während der Erstellung dieser Arbeit hat er mir im Labor für Verteilte Systeme der Fachhochschule Wiesbaden großzügig Asyl gewährt und er stand mir stets als Diskussionspartner zur Verfügung.

Zu guter Letzt danke ich den zweifellos wichtigsten Menschen meines Lebens, meiner lieben Frau Mechthild und unseren beiden Jungs, Max und Paul. Sie mussten sich während der vergangenen drei Jahre ihren Mann bzw. Vater mit dieser Arbeit mehr als nur teilen. Das wird nun anders!

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Leistungsfähigkeit eingebetteter Rechensysteme hat in den letzten Jahren einen stetigen Zuwachs erfahren. Sie verfügen heute oft über Rechenleistungen und Speicherkapazitäten, die vor nicht allzu langer Zeit noch Workstations vorbehalten waren. Auch der bisher nur im Server- und Workstation-Bereich übliche Einsatz von Mehrprozessorsystemen ist seit der Einführung der „Multicore"-Prozessoren nun auch für eingebettete Systeme realistisch, was einen weiteren Leistungsschub erwarten lässt.

Bei vielen eingebetteten Systemen ist die Menge der Betriebsmittel nicht mehr der allein kostenbestimmende Faktor: Durch einen Verzicht auf Speicher oder Rechenkapazität lässt sich oft keine wesentliche Kostenreduktion mehr erzielen. Beispielsweise finden sich heute in jedem modernen PKW oft mehr als fünfzig einzelne Steuergeräte, deren Herstellungskosten in erster Linie durch Elektronikplatine, Gehäuse, Steckverbinder, etc. bestimmt werden, während der eingesetzte Prozessor oder die Größe des Hauptspeichers in der Kostenrechnung eine eher untergeordnete Rolle spielen. Durch Verwendung moderner, leistungsfähiger Prozessoren und den entsprechenden Speicherausbau könnte die Rechenleistung solcher Geräte um ein Vielfaches gesteigert werden, während dabei ihr Preis nur geringfügig steigen würde. Um dieses Potenzial auszuschöpfen und gleichzeitig Kosten zu sparen, muss jedes dieser neuen, leistungsfähigeren Geräte die Funktionen gleich mehrerer seiner Vorgänger übernehmen, d.h. die Anzahl der Einzelgeräte muss bei gleichbleibender, bzw. steigender Gesamtfunktionalität reduziert werden. Dazu müssen Funktionen, die früher von vielen getrennten Rechnern bearbeitet wurden, nun auf wenigen gemeinsamen Rechnern zusammengeführt werden. Während ein solcher Trend in der Automobilindustrie heute erst in den Anfängen steckt ([Bro06]), ist er im Bereich der Avionik bereits in vollem Gange: Praktisch alle neueren Entwicklungen von Zivil- oder Militärflugzeugen

verwenden eine so genannte „IMA-Architektur"[1], ein System vernetzter, einheitlich aufgebauter Rechner, die als generische Plattform für eine Vielzahl von Anwendungen eingesetzt werden [SPC03, BK03]. Neben Kostenreduktion und Leistungssteigerung verspricht diese Methode aufgrund der geringeren Komplexität der Hardware auch eine höhere Zuverlässigkeit des gesamten Systems, und, da einheitliche Komponenten mehrfach verwendet werden, reduziert sich auch der Aufwand für die in der Avionik obligatorische Zertifizierung der Hardware für den Einsatz in sicherheitskritischen Anwendungen.

Die Komplexität der Software des einzelnen Rechners aber steigt dabei an. Eine Vielzahl von Programmen, die bisher auf einzelnen Prozessoren weitgehend unabhängig voneinander agiert haben, müssen sich nun einen –wenn auch leistungsfähigeren– Rechner teilen. Dadurch ergeben sich eine Reihe von Problemstellungen, die im Umfeld eingebetteter Systeme neu sind:

- *Schnittstellenvielfalt:* Die Anwendungsprogramme, die nun in einem Rechner koexistieren, haben unterschiedliche, teils sogar konträre Anforderungen an die Funktionalität ihrer unterlagerten Betriebssysteme. Die Skala der Anwendungen reicht von einfachen Programmen, die u.U. gänzlich ohne Betriebssystem arbeiten, bis hin zu Programmen, die ein vollständiges Multitasking-System mit virtueller Speicherverwaltung als Grundlage voraussetzen. Einige Programme müssen mehr oder weniger harte Zeitvorgaben erfüllen, während andere keine solchen Vorgaben haben, aber stattdessen die verfügbaren Betriebsmittel möglichst vollständig ausnutzen müssen. Es ist kaum möglich, diese Anforderungsvielfalt nur mit einer einzigen Betriebssystemschnittstelle abzudecken, und selbst wenn dies gelänge, so müssten dann alle Programme an diese neue Schnittstelle angepasst werden, was in vielen Fällen einer kompletten Neuentwicklung gleichkäme. Weitaus günstiger ist es, stattdessen die Systemarchitektur so zu gestalten, dass mehrere unterschiedliche Betriebssystemschnittstellen gleichzeitig angeboten werden können.

- *Verlust der Fehlerlokalität:* Solange Funktionen auf getrennten Rechnern liegen, gibt es –selbst wenn diese Rechner vernetzt sind– „natürliche" Grenzen der Fehlerausbreitung: Eine Fehlfunktion eines Programms, das auf einem Rechner läuft, hat nur Auswirkungen auf diejenigen Programme auf anderen Rechnern, die von der Funktionalität des fehlerhaften Programms abhängen. Eine Fehlerpropagation kann hier nur über Daten stattfinden, die zwischen den beteiligten Programmen über definierte Schnittstellen [2] ausgetauscht werden. Dadurch sind die beteiligten Programme in der Lage, an

[1] IMA steht für Integrated Modular Avionics

[2] z.B. ein Netzwerk

diesen Schnittstellen die Gültigkeit übergebener Daten zu prüfen und sich so gegen fehlerhafte Daten zu schützen. Werden nun aber mehrere, voneinander unabhängige Programme auf einen gemeinsamen Rechner gebracht, so sind ohne besondere Schutzmaßnahmen beliebige Wechselwirkungen zwischen ihnen möglich, d.h. eine Fehlfunktion eines Programms kann Fehler anderer Programme nach sich ziehen, auch wenn diese die Funktionalität des fehlerhaften Programms nicht benötigen.

Diese Probleme sind im Bereich der Server seit langem bekannt. Hier wird bereits seit Jahren die „Serverkonsolidierung" vorangetrieben, bei der es ebenfalls darum geht, bestehende und neu hinzukommende Funktionalitäten in eine einheitliche Hardware-Architektur zu integrieren. Eine solche Konsolidierung steht nun auch für eingebettete Systeme an!

Im Serverbereich wird die *Virtualisierung* als elegante Lösung der genannten Probleme eingesetzt. Dabei wird ein Modell einer Rechnerarchitektur bereitgestellt, auf dem Programme ebenso ablaufen können, wie auf einer realen Maschine. Dieses Modell wird mehrfach instanziiert, sodass in einer physischen Maschine mehrere *virtuelle Maschinen* existieren, die weitgehend unabhängig voneinander verschiedene Programme abarbeiten können. Die Implementierung solcher virtueller Maschinen geschieht mit Hilfe eines *Virtual Machine Monitors*, der die Prozessoren der realen Maschine zwischen den Kontexten der virtuellen Maschinen umschaltet.

Ursprünglich von IBM für die 360/67 Maschinen entwickelt [Cre81], hat diese Technologie heute durch ihre breite Verfügbarkeit auf Personal Computern und die Möglichkeit, unterschiedliche Betriebssysteme in einem einzigen Rechner zu betreiben, neue Aktualität erlangt.

Auch für die geforderte Konsolidierung eingebetteter Systeme bietet sich die Virtualisierung als Basistechnologie an. Allerdings gibt es hinsichtlich der Anforderungen einige wesentliche Unterschiede zu Servern und Workstations, weshalb das Konzept nicht einfach unverändert übernommen werden darf:

Echtzeitfähigkeit: Viele Anwendungen im Bereich der eingebetteten Systeme müssen in der Lage sein, „harte" oder „weiche" Zeitanforderungen zu erfüllen, d.h. bestimmte Fristen müssen entweder garantiert oder zumindest mit einer hohen Wahrscheinlichkeit eingehalten werden. Die absolute Dauer dieser Fristen ist dabei anwendungsspezifisch und kann von Fall zu Fall über mehrere Größenordnungen variieren. In Verbindung mit Virtualisierung führen diese Anforderungen zu neuen Problemstellungen: Aus der Sicht des Virtual Machine Monitors sind die virtuellen Maschinen Prozesse, zwischen denen er nach einer bestimmten Strategie umschaltet. Innerhalb der virtuellen Maschinen arbeiten Gastbetriebssysteme, die ihrerseits wiederum Prozesse verwalten. Das

heißt, die einer virtuellen Maschine durch den Virtual Machine Monitor zugewiesenen Prozessor- und Rechenzeitkontingente werden anhand einer weiteren, dann für die virtuelle Maschine lokalen Strategie, auf Anwendungsprogramme verteilt. Hier überlagern sich also zwei Strategien, die letztendlich beide gemeinsam das Zeitverhalten der Programme bestimmen.

Mehrprozessorfähigkeit: Die stetige Erhöhung des Prozessortaktes, die in den vergangenen Jahren zu einem Anstieg der Prozessorleistung geführt hat, stößt mittlerweile an physikalische Grenzen. Mit der etablieren Halbleitertechnologie kann in Zukunft auf diesem Wege kein weiterer Leistungszuwachs mehr erzielt werden. Neuere Prozessoren werden daher dem Bedarf nach mehr Rechenleistung durch Multicore-Architekturen gerecht, bei denen es sich im Wesentlichen um auf einem Chip integrierte Mehrprozessorsysteme handelt. Bei Desktop-Rechnern sind bereits heute Zwei- und Vierkern-Prozessoren etablierter Standard. Für eingebettete Systeme zeichnet sich ein ähnlicher Trend ab. Um das Leistungspotenzial eines solchen Mehrprozessorsystems auszuschöpfen, müssen die anstehenden Aufgaben auf die zur Verfügung stehenden Prozessoren verteilt und von diesen parallel abgearbeitet werden. Im Serverbereich besteht die typische Rechenlast aus einer großen Zahl voneinander unabhängig agierender Prozesse. Traditionell wird hier die parallele Verarbeitung üblicherweise dadurch erreicht, dass die Prozesse dynamisch den verschiedenen Prozessoren zugewiesen werden, wobei jedoch der einzelne Prozess weiterhin sequenziell abgearbeitet wird. Erst seit neuerer Zeit besteht auch eine Tendenz zur Parallelisierung einzelner Prozesse. Bei eingebetteten Systemen ist von einer eher kleineren Anzahl von Prozessen auszugehen. Deshalb genügt es hier vielfach nicht, die Prozessorzuteilung allein auf der Ebene der Prozesse vorzunehmen, sondern zumindest einige Prozesse müssen in sich parallelisiert werden, d.h. ihre Rechenlast muss in mehrere, dann parallel abzuarbeitende Teilprozesse aufgeteilt werden. Dabei kann die Kommunikation der Teilprozesse untereinander sowohl über gemeinsamen Speicher, als auch über den Austausch von Nachrichten erfolgen. Diese Aufgabe betrifft zunächst die Anwendungsprogrammierer, die ihre Algorithmen entsprechend implementieren müssen, sowie die Entwickler von Compiler-Werkzeugen, die eine automatische oder halbautomatische Generierung von parallelem Code unterstützen müssen. Aber unabhängig von der verwendeten Programmiermethodik und den eingesetzten Werkzeugen stellen solche parallelen Prozesse gegenüber dem unterlagerten Betriebssystem –insbesondere dann, wenn sie Zeitanforderungen unterliegen– eine neue Art der Rechenlast dar, die von Server- und Desktopsystemen bisher nicht ausreichend berücksichtigt wurde. Das Betriebssystem und die ihm unterlagerte Virtualisierungsschicht müssen geeignete Mechanismen zur effizienten Unterstützung solcher paralleler Prozesse bieten.

Statische Konfiguration: Im Server- und Workstation-Bereich müssen Rechensysteme ein hohes Maß an dynamischer Rekonfigurierbarkeit aufweisen: Betriebsmittel müssen, je nach aktueller Lastanforderung, verschiedenen virtuellen Maschinen zur Laufzeit zugewiesen bzw. ihnen entzogen werden können, und neue virtuelle Maschinen müssen dynamisch erzeugt und wieder entfernt werden können. Auf diesem Gebiet dürfen für eingebettete Systeme Abstriche gemacht werden: Die zu erwartende Lastsituation ist hier wesentlich genauer vorab bekannt als bei Servern oder Workstations. Mitunter ist eine starre Betriebsmittelzuteilung sogar ausdrücklich erwünscht: Beim Einsatz in sicherheitskritischen Anwendungen muss die Integrität jeder virtuellen Maschine unabhängig von anderen virtuellen Maschinen gezeigt werden. Eine solche Argumentation wird durch eine feste Betriebsmittelzuteilung, die zur Laufzeit nicht änderbar ist, erheblich vereinfacht.

Die speziellen Anforderungen an eine Virtualisierung für eingebettete Systeme werden von den bisher existierenden Implementierungen dieser Technik nicht ausreichend abgedeckt. Dies ist jedoch keine grundsätzliche Einschränkung der Virtualisierung an sich, sondern einfach darin begründet, dass Virtualisierung bisher ausschließlich im Bereich der Server und Workstations eingesetzt, und demzufolge auf die Bedürfnisse dieses Bereiches zugeschnitten wurde.

Das Ziel dieser Arbeit ist nun die Erforschung der Anwendbarkeit und die Anwendung von Virtualisierung auf eingebettete Systeme mit mehreren Prozessoren. Bestehende Schnittstellen zur Virtualisierung werden im Hinblick auf ihre Echtzeitfähigkeit untersucht, und es werden neue Schnittstellen zur Virtualisierung auf Mehrprozessormaschinen geschaffen und erprobt, die die genannten spezifischen Anforderungen eingebetteter Systeme berücksichtigen. Dabei soll es insbesondere ermöglicht werden, dass Programme mit unterschiedlich harten Zeitanforderungen und unterschiedlichen Fristen effizient und sicher in getrennten virtuellen Maschinen auf einem gemeinsamen Rechner koexistieren können.

Diese Themenstellung berührt drei bekannte Gebiete der Informationstechnologie:

1. Virtualisierung
2. Echtzeitverarbeitung
3. Mehrprozessorsysteme

Von diesen drei Gebieten ist keines wirklich neu. Allerdings ergeben sich aus ihrer gemeinsamen Betrachtung zahlreiche neue Fragestellungen, die es im Rahmen dieser Arbeit zu untersuchen gilt:

Anwendbarkeit klassischer Planungsverfahren: Ziel einer Virtualisierungsumgebung ist es, jedem Subsystem, das in einer virtuellen Maschine arbeitet, den Eindruck zu vermitteln, ihm stünde eine vollständige physische Maschine zur Verfügung. In [PG74] fordern Popek und Goldberg, dass es für ein Programm nicht erkennbar sein darf, ob es in einer virtuellen oder einer physischen Umgebung läuft. Dabei nehmen die Autoren jedoch das Zeitverhalten ausdrücklich von ihren Betrachtungen aus, denn eine Veränderung des Zeitverhaltens der Gastsysteme durch die Virtualisierung ist nicht zu vermeiden. Somit ist zunächst die Frage zu beantworten, wie sich diese Veränderung des Zeitverhaltens auf Echtzeitanwendungen auswirkt: Es ist zu prüfen, ob die bekannten Methoden der Echtzeitplanung in einer Virtualisierungsumgebung weiterhin anwendbar bleiben, und, falls dem so ist, welche prinzipiellen Einschränkungen dabei gelten.

Verbesserungsmöglichkeiten durch spezialisierte Planungsverfahren: Da es eine Vielfalt von Anwendungen mit stark unterschiedlichen Zeitanforderungen in einem System zusammenzuführen gilt, ist davon auszugehen, dass die sich ergebendenen Einschränkungen beim Einsatz klassischer Planungsverfahren zumindest für einen Teil der Anwendungen nicht akzeptabel sein werden. Für diese Gruppe von Anwendungen muss nach alternativen Planungsverfahren gesucht werden, die in Kenntnis der Virtualisierungsschicht und möglicherweise unter Inkaufnahme anderer, für den jeweiligen Anwendungsfall akzeptabler Einschränkungen die Anwendungsanforderungen erfüllen können. Diese neuen Planungsverfahren müssen jedoch mit den klassischen Verfahren koexistieren können, damit diese auch weiterhin anwendbar bleiben.

In bestehenden Virtualisierungssystemen sind gewöhnlich keine Möglichkeiten der Kommunikation zwischen Gastsystemen und dem unterlagerten Virtual Machine Monitor vorgesehen (schließlich soll die Virtualisierung für das Gastsystem unsichtbar bleiben). Spezialisierte Planungsverfahren könnten hier neue Wege gehen, z.B. indem Gastsysteme mit den Umschaltungen des Virtual Machine Monitors synchronisiert werden oder indem der Virtual Machine Monitor auf Anforderung von Gastsystemen dynamisch Betriebsmittel zuweist. Die spezialisierten Planungsverfahren würden somit geeignete Kommunikationsmechanismen zwischen dem Scheduler eines Virtual Machine Monitors und den Schedulern der Gastsysteme nutzen, um bessere Echtzeiteigenschaften zu erzielen. Solche Kommunikationsmechanismen gilt es zu definieren und anhand einer prototypischen Implementierung zur evaluieren.

Nutzbarkeit der Mehrprozessorarchitektur: Bei der Konsolidierung eingebetteter Systeme kann der durch eine Mehrprozessorarchitektur gegebene Zuwachs an Rechenleistung zunächst dazu genutzt werden, mehr Rechenlast in Form von mehr Prozessen zu tragen. Dies ist auch notwendig, wenn –wie

beschrieben– ein einzelner, leistungsfähiger Rechner die Aufgaben mehrerer kleinerer Rechner übernehmen soll. Im Echtzeitumfeld kann das Mehr an Rechenleistung aber auch zur Verbesserung des Zeitverhaltens genutzt werden. So können kürzere Latenzzeiten erzielt oder kürzere Fristen eingehalten werden. Hierzu müssen die zeitkritischen Aufgaben auf mehrere Teilprozesse verteilt werden, die dann parallel von mehreren Prozessoren abgearbeitet werden. Es existieren spezielle Programmiermodelle, die eine solche Aufteilung einer Rechenlast auf mehrere Teilprozesse ermöglichen. Diese Modelle und die zugehörigen Werkzeuge sind zwar ausdrücklich *nicht* Thema dieser Arbeit, jedoch bedürfen die von ihnen erzeugten parallelen Programme der Unterstützung auf der Betriebssystemebene. Soll beispielsweise eine gegebene Aufgabe innerhalb einer kürzeren Frist abgearbeitet werden, so genügt es nicht allein, den betreffenden Prozess in parallele Teilprozesse zu zerlegen: die Bearbeitung der Gesamtaufgabe beginnt mit dem Start des ersten Teilprozesses und sie endet erst, wenn der letzte der Teilprozesse seine Arbeit verrichtet hat. Daher muss ein Betriebssystem sicherstellen, dass alle Teilprozesse gleichzeitig auf den verfügbaren Prozessoren ausgeführt werden, damit alle gleichzeitig und möglichst früh beendet werden können.

Für nicht-parallelisierte, bzw. nicht-parallelisierbare Echtzeitprozesse gibt es eine vollkommen andere Möglichkeit, wie eine Mehrprozessorarchitektur vorteilhaft genutzt werden kann: Echtzeitprozesse, deren Zeitanforderungen miteinander in Konflikt stehen, können auf jeweils eigenen Prozessoren ausgeführt werden. Dadurch werden die auf einem Einprozessorsystem bestehenden Konflikte vermieden, was für jeden der Echtzeitprozesse zu verbesserten Echtzeit-Eigenschaften führt.

Beide Nutzungsmöglichkeiten gilt es zu untersuchen, und nach Möglichkeit sollten beide Varianten gleichzeitig in einem System anwendbar sein.

Schaffung einer geeigneten Infrastruktur: Technisches Ziel dieser Arbeit ist die Modellierung, der Entwurf, die prototypische Implementierung und die Bewertung einer Scheduler-Infrastruktur für Virtualisierungsumgebungen, auf deren Basis die vorgenannten Ziele erreicht werden können, d.h. es sollen sowohl einzelne als auch parallele Prozesse mit mehr oder weniger hohen Echtzeitanforderungen sicher in einem einzigen Rechner koexistieren können. Dabei sollen Echtzeitgastsysteme in der Lage sein, Planbarkeitsanalysen durchzuführen und ggf. Garantien über die fristgerechte Ausführung ihrer Anwendungen abzugeben, ohne dabei Kenntnisse über andere, im selben Rechensystem existierende Gastsysteme besitzen zu müssen. Gleichzeitig sollen Nicht-Echtzeitgastsysteme durch Nutzung von Rechenkapazitäten, die in der Echtzeitplanung nicht ausgeschöpft werden können, für eine gute Auslastung des Rechensystems sorgen. Die zu betrachtenden Arten von Rechenlasten orien-

tieren sich an den typischen Anwendungen eingebetteter Systeme, d.h. die zu erfüllenden Zeitanforderungen können über sehr weite Bereiche variieren, jedoch sind sie vorab bekannt. Somit werden keine sehr hohen Anforderungen an eine dynamische Konfigurierbarkeit des Systems gestellt, und insbesondere können Anwendungen mit Echtzeitanforderungen statisch geplant werden. Die zu entwerfende Infrastruktur soll einem Systemkonfigurator also lediglich Mechanismen anbieten, mit deren Hilfe konkurrierende Anforderungen verschiedener Anwendungen im Rahmen einer statischen Konfiguration befriedigt werden können.

Der Rest dieser Arbeit ist folgendermaßen aufgeteilt: Kapitel 2 stellt Grundlagen zu den genannten drei Gebieten zusammen. In Kapitel 3 werden eigene Modelle zur Beschreibung von Prozesshierarchien und zur Echtzeitplanung auf den Ebenen dieser Hierarchien, zur Abschätzung der Laufzeitkosten der Virtualisierung und zur Beschreibung von Ein- und Mehrprozessorbetriebssystemen in Virtualisierungsumgebungen entwickelt. In Kapitel 4 werden unter Verwendung dieser Modelle die sich aus der jeweils paarweisen Kombination der drei Themengebiete ergebenden Fragestellungen analysiert. Dies führt am Ende des Kapitels zu einer Anforderungsbeschreibung für eine Infrastruktur zur Prozesssteuerung auf der Ebene des Virtual Machine Monitors. Diese Infrastruktur ermöglicht die Koexistenz von Gastsystemen unterschiedlichster Klassen in einem gemeinsamen Rechensystem. Kapitel 5 skizziert zunächst eine Scheduler-Infrastruktur die die Anforderungen aus Kapitel 4 erfüllt. Im letzten Abschnitt dieses Kapitels wird eine konkrete Implementierung dieser Infrastuktur für den Virtual Machine Monitor „Xen" beschrieben. Kapitel 6 liefert eine Bewertung der mit dieser Implementierung praktisch erzielten Ergebnisse. In Kapitel 7 findet sich eine abschließende Zusammenfassung der Arbeit.

Kapitel 2

Grundlagen

In dieser Arbeit geht es um die Anwendung von Schnittstellen zur Virtualisierung auf Mehrprozessorsystemen, auf denen Programme arbeiten, die Echtzeitanforderungen unterliegen. Dieses Kapitel stellt die nötigen Grundlagen zu den drei Kerngebieten Mehrprozessorsysteme, Echtzeitverarbeitung und Virtualisierung zusammen.

2.1 Mehrprozessorsysteme

Eine von Flynn eingeführte Klassifikation ([Fly72]) ordnet Rechensysteme danach, ob Daten und/oder Instruktionen parallel abgearbeitet werden. Dabei ergeben sich vier mögliche Kombinationen:

1. **SISD** (engl.: *Single instruction single data stream*): Ein einzelner Prozessor verarbeitet einen einzelnen Befehlsstrom einen einzelnen Datenstrom. Dies ist der klassische Einprozessor-Fall.

2. **SIMD** (engl.: *Single instruction multiple data stream*): Eine einzelner Maschinenbefehl bewirkt die gleichzeitige Verarbeitung mehrerer Datenelemente. D.h. ein und dasselbe Programm verarbeitet gleichzeitig mehrere Datensätze nach dem gleichen Algorithmus.

3. **MISD** (engl.: *Multiple instruction single data stream*): Mehrere, verschiedene Instruktionen arbeiten gleichzeitig auf einem gemeinsamen Datenstrom.

4. **MIMD** (engl.: *Multiple instruction multiple data stream*): Mehrere Prozessoren führen verschiedene Befehlsströme (Programme) aus und verarbeiten dabei verschiedene Datenströme.

MISD-Systeme werden hier nur aus Gründen der Systematik angeführt, ein solches System wurde nie implementiert[1]. Im Gegensatz dazu haben SIMD-Systeme praktische Anwendung im Bereich der Vektorrechner gefunden. Sie eignen sich für die Bearbeitung sehr spezifischer Problemstellungen, sind aber für die im Rahmen dieser Arbeit betrachteten Aufgaben nicht geeignet. In dieser Arbeit werden ausschließlich SISD- und MIMD-Systeme betrachtet. Im Rest der Arbeit werden erstere als „Einprozessorsysteme", letztere je nach der Enge der Kopplung zwischen den Prozessoren als „Mehrprozessorsysteme" oder „Mehrrechnersysteme" bezeichnet.

Allgemein versteht man unter „Mehrprozessorsystemen" Rechensysteme mit mehreren Verarbeitungseinheiten (Prozessoren), die einen einzigen, gemeinsamen Adressraum nutzen (vgl. [PH05], S. 512). Es existiert eine Vielzahl möglicher Architekturen, die sich in der Art der verwendeten Prozessoren, der Art der Kopplung zwischen den Prozessoren und der Art der Speicherankopplung unterscheiden. Im Folgenden werden zunächst einige Klassifikationen von Mehrprozessorsystemen anhand der genannten Kriterien angeführt. Anschließend folgen einige Grundsatzüberlegungen zu der durch Parallelverarbeitung erzielbaren Leistungssteigerung.

2.1.1 Klassifikation von Mehrprozessorarchitekturen

Mehrprozessorsysteme lassen sich, wie eingangs dieses Abschnitts gesagt, nach verschiedenen Kriterien weiter klassifizieren:

Klassifikation nach Homogenität

Je nachdem, ob die eingesetzten Prozessoren gleich- oder verschiedenartig aufgebaut sind, spricht man von *homogenen* oder *heterogenen* Mehrprozessorsystemen. Die Multicore-Prozessoren, die für diese Arbeit von besonderem Interesse sind, verwenden in der Mehrzahl mehrere gleichartige Prozessorkerne, zählen also zu den homogenen Mehrprozessorarchitekturen. Eine Ausnahme bildet hier jedoch der „Cell"-Prozessor ([Kah05]), der zwei unterschiedlich aufgebaute Arten von Prozessorkernen auf einem Chip besitzt.

[1] Mit einer gewissen Berechtigung könnte man das „Pipelining", mit dessen Hilfe heutige Prozessoren intern mehrere Befehle gleichzeitig abarbeiten, als MISD-Eigenschaft auffassen. Auf der Programmierebene tritt diese Eigenschaft jedoch nicht in Erscheinung

Klassifikation anhand der Speicherankopplung

Mehrprozessor-Architekturen lassen sich, abhängig von der Art der Kopplung zwischen Prozessoren und Speicher in grob zwei Kategorien einteilen ([PH05], S. 528):

- *Shared memory:* Eine Mehrprozessor-Architektur, bei der alle Prozessoren direkten Zugriff auf einen gemeinsamen Speicher haben, wird als „Shared-Memory-Architektur" bezeichnet. Je nach Art der Speicheraufteilung lassen sich diese Architekturen weiter unterteilen ([YATR+87]):
  - UMA: *Uniform memory access*: Unter „UMA-Systemen" versteht man Mehrprozessorarchitekturen, bei denen alle Prozessoren über einen einheitlichen Pfad (meist einen gemeinsamen Bus) auf einen globalen Hauptspeicher zugreifen. Die Geschwindigkeit des Zugriffs auf den Hauptspeicher ist also für alle Prozessoren gleich (Siehe Abbildung 2.1).
  - NUMA: *Non-uniform memory access*: Im Gegensatz zu UMA-Systemen gibt es bei „NUMA-Systemen" Prozessor-lokale Speicher (Siehe Abbildung 2.2). Zwar kann auch hier jeder Prozessor auf jede Speicherzelle des Systems zugreifen, jedoch ist die Zugriffsgeschwindigkeit stark davon abhängig, ob ein Prozessor auf den ihm zugeordneten, lokalen Speicher, oder auf den (lokalen) Speicher eines anderen Prozessors zugreift.
- NoRMA: *No remote memory access*: Als „NoRMA-System" oder auch „Cluster" wird eine Anzahl Rechnerknoten bezeichnet, die von außen als ein einziger Computer angesehen werden können, bei denen aber die einzelnen Knoten keinen direkten Zugriff auf den Speicher der anderen Knoten haben. In der Regel sind die Knoten über ein schnelles Netzwerk miteinander verbunden.

Klassifikation anhand der Symmetrie

Ist eine Shared-Memory-Architektur darüber hinaus homogen (s.o.) und ist der Speicherzugriff symmetrisch für alle Prozessoren, d.h. adressiert jeder Prozessor mit derselben physikalischen Adresse dieselbe Speicherzelle, so spricht man auch von einer „SMP-Architektur" (engl.: *symmetric multiprocessor*).

Da bei SMP-Systemen die Speicheradressierung für alle Prozessoren gleich ist, werden Prozesse durch Daten im gemeinsamen Speicher repräsentiert und sind an

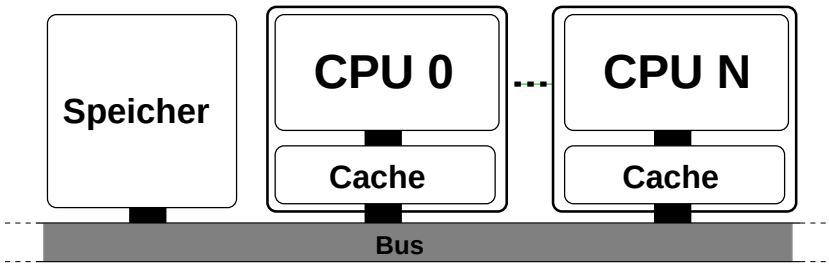

Abbildung 2.1: UMA-Architektur (mit Prozessor-lokalen Caches).

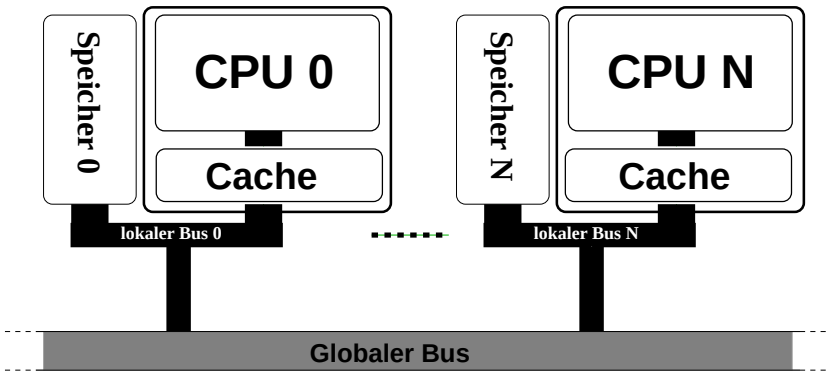

Abbildung 2.2: NUMA-Architektur.

keinen bestimmten Prozessor gebunden. Sie können dynamisch auf die Prozessoren verteilt (und bei Bedarf auch jederzeit neu verteilt) werden. In aller Regel sind SMP-Systeme zugleich auch UMA-Systeme. Zwar kann auch eine NUMA-Architektur prinzipiell so aufgebaut sein, dass der Speicherzugriff für alle Prozessoren symmetrisch ist, aber für die Nutzung der speziellen Vorteile einer NUMA-Architektur ist es erforderlich, dass die Daten eines Prozesses „nahe bei" dem betreffenden Prozess, d.h. im lokalen Speicher desjenigen Prozessors liegen, der den Prozess ausführt. Somit gibt es bei NUMA-Systemen gute Gründe, Prozesse an bestimmte Prozessoren zu binden. Prozessverlagerungen sind zwar technisch möglich, aber sie sind mit erheblich höheren Laufzeitkosten verbunden, als bei UMA-Systemen.

Bei UMA-Architekturen entwickelt sich der gemeinsame Bus bei steigender Prozessorzahl sehr schnell zum leistungsbegrenzenden Faktor([ZA95]). Deshalb verfügen die Prozessoren heutiger UMA-Systeme fast immer über lokale Cache-Speicher, die den Bus entlasten. Aufgrund dieser lokalen Speicher aber zeigen diese Systeme nun auch NUMA-typische Eigenschaften: Der Zugriff auf nicht-lokale, d.h. nicht im Cache befindliche Daten ist verhältnismäßig teuer und die Verlagerung eines Prozesses von einem Prozessor zu einem anderen ist mit Kos-

ten verbunden, da die Cache-Inhalte des bisherigen Prozessors dabei nicht „mitgenommen" werden können. Da die Caches Kopien der Daten aus dem Arbeitsspeicher enthalten, ist es nötig, diese „kohärent" zu halten. Wenn beispielsweise mehrere Prozessoren dasselbe Datum in ihren Cache kopiert haben, müsste, sobald einer der Prozessoren dieses Datum verändert, der geänderte Wert in den Hauptspeicher kopiert werden und die übrigen Prozessoren müssen beim nächsten Zugriff ihre Kopie des Datums verwerfen und den neuen Wert aus dem Hauptspeicher beziehen. Da diese Methode wegen der verhältnismäßig langen Speicherzugriffszeit zu großen Performance-Einbrüchen führen würde, wird in der Regel das „MESI"-Protokoll (engl.: *Modified Exclusive Shared Invalid*) verwendet (siehe [CSG98]): Dabei führt die Änderung eines im Cache befindlichen Datums nicht zwangsläufig sofort zu einem Zurückschreiben in den Hauptspeicher, sondern der Cache, der das geänderte Datum enthält, kann dieses auf Anfrage direkt an die Caches der übrigen Prozessoren liefern.

Diese Operationen werden in der Regel durch eine geeignete Logik sichergestellt, sodass –abgesehen von Zugriffszeiten– die Caches aus Sicht des Programmierers transparent sind. Auch die Prozessoren von NUMA-Systemen sind in der Regel mit Caches ausgestattet. Dadurch ergibt sich auch hier das Problem der Cache-Kohärenz, das die meisten existierenden NUMA-Systeme ebenfalls mit Hilfe einer entsprechenden Logik lösen. Man bezeichnet solche Systeme auch als "ccNUMA"-Systeme (engl.: *cache coherent NUMA*). Mit Hilfe der DSM-Technik (engl.: *distributed shared memory*) lassen sich in einem NoRMA-System lokale Speicherinhalte auch auf entfernten Knoten transparent replizieren ([FR86, FBYR88, CH05]). Damit kann ein (homogenes) NoRMA System –zumindest aus Sicht der Anwenderprogramme– prinzipiell in ein NUMA-System transformiert werden, wobei aber der Zugriff auf nicht-lokalen Speicher nochmals wesentlich teurer als bei einer „echten" NUMA-Architektur ist. Daher verbieten sich –zumindest im Echtzeitumfeld– Prozessmigrationen zur Laufzeit bei NoRMA-Systemen in der Regel.

Für diese Arbeit sind in erster Linie Multicore-Prozessoren relevant, bei denen es sich in der Regel um SMP-Systeme mit UMA-Architektur handelt. Deshalb werden hier primär solche Systeme betrachtet. Jedoch wird sich zeigen, dass einige der vorgestellten Konzepte auch auf NUMA- und, unter bestimmten Voraussetzungen, sogar auf NoRMA-Architekturen und heterogene Mehrprozessorsysteme anwendbar sind.

2.1.2 Chip-interne Parallelität: SMT- und Multicore-Prozessoren

Bis vor einigen Jahren wurden stets mehrere, separate Prozessorchips zum Aufbau von Mehrprozessorsystemen verwendet, sodass diese schon äußerlich unmittelbar als solche erkennbar waren. Dabei wurden auf dem Chip neben dem eigentlichen Prozessorkern teilweise weitere Baugruppen wie Speicheransteuerung, Caches, Speicherverwaltung (MMU), Gleitkommaprozessor, etc. mit integriert. Um die Leistungsfähigkeit der Prozessoren zu steigern, wurden –neben einer stetigen Erhöhung der Taktfrequenz– Techniken wie „Pipelining" und „Mehrfachzuordnung" eingeführt, die das gleichzeitige Abarbeiten mehrerer Befehle ermöglichen (vgl. [PH05], Kap. 6). Diese Maßnahmen stellen bereits eine Form der Chip-internen Parallelverarbeitung dar, die allerdings für den Programmierer unsichtbar bleibt, da nach wie vor ein einziger sequenzieller Strom von Befehlen abgearbeitet wird. Problematisch wird diese sogenannte .‚feingranulare Parallelität" [1] dann, wenn ein Befehl auf das Ergebnis eines vorangegangenen, aber noch in Bearbeitung befindlichen Befehls Bezug nimmt: Die Bearbeitung des betroffenen Befehls muss dann so lange angehalten werden, bis das erforderliche Ergebnis vorliegt. Solche „Pipeline-Hemmnisse" haben dementsprechend einen erheblichen Durchsatzverlust zur Folge. Sie werden mit wachsender Anzahl parallel abzuarbeitender Instruktionen zunehmend häufiger: Auch bei optimaler Codierung haben sequenzielle Programme oft nur eine begrenzte Anzahl aufeinander folgender Befehle, die voneinander unabhängig sind.

Etwa um das Jahr 2000 kamen „SMT-Prozessoren" auf (engl.: *Symmetric multithreading*). Diese, auch als „hyperthreading-Prozessoren" bezeichneten Chips stellen sich äußerlich als einzelne Prozessoren dar, besitzen aber intern mehrere Registersätze (insbesondere auch mehrere Programmzähler), die aus Programmsicht als eigenständige Prozessorkontexte angesehen werden können. Damit wird die Vorstellung eines einzigen abzuarbeitenden Befehlsstromes aufgegeben: SMT-Prozessoren führen, wie „echte" Mehrprozessorsysteme, mehrere Programmpfade gleichzeitig aus. Abgesehen von den Registern sind aber bei einem SMT-Prozessor die meisten internen Komponenten des Prozessorkerns, insbesondere die Arithmetisch-logische Einheit (ALU) und die Gleitkommaeinheit (FPU), nur einfach vorhanden, sodass die verschiedenen Programmausführungen nicht gleichzeitig darauf zugreifen können. Werden auf mehreren Pfaden gleichzeitig Befehle ausgeführt, die die gleiche interne Komponente benötigen, so sorgt eine interne Arbitrierungslogik des Chips dafür, dass diese Befehle nacheinander ausgeführt werden. Durch solche internen Zugriffskonflikte behindern sich die gleichzeitig ausgeführten Programme gegenseitig. Dass SMT-Prozessoren dennoch einen Leistungszuwachs gegenüber Einprozessorsystemen aufweisen kön-

[1] Auch: Parallelität auf Instruktionsebene, engl. *instruction level parallelism*

nen, rührt daher, dass die Komponenten des Prozessorkerns bei der herkömmlichen, einfädigen Programmausführung häufig nur zeitweise genutzt werden, sodass sich mit mehreren, gleichzeitig ausgeführten Programmpfaden eine bessere durchschnittliche Auslastung dieser Betriebsmittel ergibt (vgl. [PH05], S. 537). Beispiele für SMT-Prozessoren sind der Pentium 4 und der Xeon-Prozessor von Intel. Der MIPS-34k Prozessor von MIPS Technologies, der wegen seiner geringen Leistungsaufnahme für eingebettete Systeme besonders geeignet ist, enthält insgesamt fünf Prozessorkontexte, von denen aber zwei bzw. drei jeweils eine gemeinsame Speicherverwaltungseinheit (MMU) besitzen, sodass die ihnen zugeordneten Programme stets nur gemeinsame Adressräume verwenden können. Bei den SMT-Prozessoren von Intel hingegen besitzt jeder Prozessorkontext eine eigene MMU.

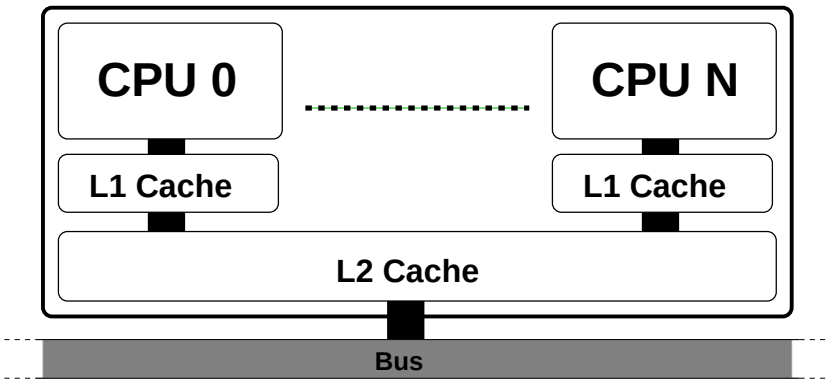

Abbildung 2.3: Multicore-Prozessor mit gemeinsamem Level-2 Cache.

Im Gegensatz zu SMT-Prozessoren enthalten die um 2004 erschienenen „Multicore-Prozessoren" mehrere, voneinander unabhängige Prozessorkerne auf einem gemeinsamen Chip. Obwohl diese Chips, wie die SMT-Prozessoren, sich bezüglich ihrer äußeren Schnittstelle als einzelne Prozessoren darstellen, sind sie funktional eher mit aus diskreten Prozessoren aufgebauten Mehrprozessorsystemen vergleichbar. Lediglich ein Teil des auf dem Chip befindlichen Cache wird bei den heute eingesetzten Multicore-Prozessoren in der Regel von mehreren Prozessorkernen gemeinsam genutzt. Abbildung 2.3 zeigt das Prinzipbild eines Multicore-Prozessors mit N Prozessorkernen, bei dem der Level-2 Cache von allen Prozessorkernen gemeinsam genutzt wird, während jeder Kern einen eigenen Level-1 Cache besitzt. Typische Vertreter der Multicore-Prozessoren sind die „Core 2 Duo" Prozessoren von Intel oder AMDs „Opteron"

SMP- und SMT-Eigenschaften lassen sich miteinander kombinieren: So gibt es sowohl Einzelprozessoren als auch Multicore-Prozessoren, deren Prozessorkerne intern mehrere Registerkontexte bereitstellen, d.h. die wiederum SMT-

Prozessoren sind (vgl. [AR06], S. 259).

2.1.3 Leistungssteigerung durch Einsatz mehrerer Prozessoren

Alle bekannten Maßnahmen zur Steigerung der Rechenleistung eines einzelnen Prozessors stoßen früher oder später an unüberwindliche Grenzen: Die Erhöhung der Taktfrequenz führt zu einer Erhöhung der Leistungsaufnahme des Prozessors, Pipeline-Hemmnisse nehmen bei wachsender Anzahl gleichzeitig bearbeiteter Befehle überproportional zu. Mit einzelnen Prozessoren lässt sich keine beliebig skalierbare Rechenleistung erzielen.

Dagegen verspricht der Einsatz mehrerer, parallel arbeitender Prozessoren bei naiver Betrachtungsweise zunächst eine lineare Skalierbarkeit: benötigt ein Prozessor zur Bearbeitung einer gegebenen Aufgabe eine Zeitdauer T, so müssten N Prozessoren die gleiche Aufgabe in der Zeit $\frac{T}{N}$ erledigen können.

Tatsächlich werden Mehrprozessorsysteme heute in der Regel zum Zweck der Leistungssteigerung durch das gleichzeitige (d.h. parallele) Abarbeiten mehrerer Programme, bzw. Programmteile eingesetzt. In der Realität ist eine lineare Skalierbarkeit der Rechenleistung aber nur selten gegeben. Dafür gibt es mehrere Gründe:

- *Nicht-Parallelisierbarkeit von Algorithmen:* Um die Rechenlast einer Aufgabe gleichmäßig auf mehrere Prozessoren zu verteilen, muss das betreffende Programm parallelisiert werden, d.h. es muss in mehrere Teilprogramme zerlegt werden, die jeweils einen Teil der anstehenden Aufgabe bewältigen. Eine solche Parallelisierung erfordert in der Regel ein manuelles Umschreiben des Programms, und nicht jeder Algorithmus ist dafür gleichermaßen geeignet: während etwa Feldoperationen sehr gut „skalieren" ist dies z.B. für Ein/Ausgabeoperationen nicht der Fall. Deshalb haben nahezu alle parallelen Programme stets mehr oder weniger stark ausgeprägte nichtparallele (*sequenzielle*) Anteile.

  Das „Gesetz von Amdahl" ([Amd67]) besagt allgemein, dass die durch eine bestimmte Verbesserung mögliche Leistungssteigerung durch den Umfang der Verwendung der verbesserten Komponente bestimmt ist. Angewandt auf die Parallelverarbeitung mit Mehrprozessorsystemen, bedeutet dies, dass bei einem Programm, dessen Code zu einem Anteil P parallel und zu einem Anteil S sequenziell ausgeführt werden kann[1], der durch parallele Ausfüh-

[1] D.h. $P + S = 1$

rung dieses Programms auf N Prozessoren erzielbare Leistungsgewinn G gleich:

$$G = \frac{P+S}{\frac{P}{N}+S} = \frac{1}{\frac{1-S}{N}+S} \tag{2.1}$$

ist. Nur für $S = 0$ (und damit: $P = 1$) ist dieser Wert gleich N, für alle Werte von $S > 0$ ist er kleiner, d.h. eine lineare Leistungssteigerung kann nur für Programme erreicht werden, die keinerlei sequenziellen Anteil besitzen. Der Einfluss des sequenziellen Anteils auf die erreichbare Leistungssteigerung nimmt mit zunehmender Prozessorzahl noch weiter zu: Während ein Programm mit 20% sequenziellem Anteil bei zwei Prozessoren immerhin 83% der möglichen Leistungssteigerung erfahren kann, sind es bei 16 Prozessoren nur noch 25%.

- *Kommunikationsaufwand:* Die Anteile eines parallelen Programms haben in der Regel Kommunikationsbedarf, d.h. sie kommunizieren, beispielsweise über Nachrichten, miteinander. Dies bedeutet einen zusätzlichen Aufwand, der bei einem Einprozessorsystem nicht besteht. In [Ous80] argumentiert Ousterhout, dass bei steigender Prozessorzahl der Aufwand für die Kommunikation zwischen den Prozessoren letztendlich immer dominant wird, da die Anzahl der zur Verbindung der Prozessoren miteinander nötigen Kommunikationskanäle schneller steigt, als die Anzahl der Prozessoren selbst. Je nach Umfang der erforderlichen Kommunikation wird also bei steigender Prozessorzahl früher oder später ein Punkt erreicht, ab dem der Mehraufwand für die Kommunikation den Zuwachs an Rechenleistung überwiegt.

- *Gemeinsame Betriebsmittel:* Kommunizieren die Anteile eines Prozesses statt über Nachrichten über den Austausch gemeinsamer Daten, so müssen die Prozessoren über einen gemeinsamen Bus auf diese Daten zugreifen. Dabei entwickelt sich dieser Bus mit steigender Teilnehmerzahl sehr schnell zum leistungsbegrenzenden „Flaschenhals" [ZA95]. Durch das Einführen Prozessor-lokaler Speicher (z.B. Caches) lässt sich das Problem zwar abmildern, aber nicht beseitigen. Letztendlich sind auch gemeinsame Daten nichts anderes als Kommunikationskanäle, die –nach dem oben Gesagten– früher oder später das Ergebnis dominieren.

2.2 Echtzeitverarbeitung

Bei der Konsolidierung eingebetteter Systeme werden Aufgaben auf einem Rechner zusammengefasst, die hinsichtlich ihres Zeitverhaltens unterschiedliche –teils

sogar widersprüchliche– Anforderungen zu erfüllen haben. Es gibt Aufgaben mit:

- „harten" Echtzeitanforderungen, bei denen Prozesse innerhalb einer vorgegebenen Frist zum Ergebnis kommen müssen. Jegliches Überschreiten der Frist wird als fataler Fehler gewertet.
- „weichen" Echtzeitanforderungen, bei denen ebenfalls generell die Einhaltung einer gegebenen Frist erwartet wird, wobei aber eine Überschreitung nicht als Fehler, sondern als Wertminderung des Ergebnisses aufgefasst wird, d.h. eine Fristverletzung kann toleriert werden, solange sie selten genug auftritt und solange die Abweichung nicht zu groß wird.
- keinerlei Zeitanforderungen. Von Prozessen dieser Kategorie wird erwartet, dass sie die verfügbaren Betriebsmittel möglichst vollständig ausschöpfen und so die Prozessorauslastung maximieren. Da sie keinen festen Fristen unterliegen, darf ihre Verarbeitungszeit abhängig von den aktuell verfügbaren Betriebsmitteln variieren. Arbeiten mehrere solcher Programme auf gemeinsamen Betriebsmitteln, so wird erwartet, dass sie diese jeweils anteilig nutzen.

Unabhängig davon, ob es sich um harte oder weiche Zeitanforderungen handelt, hängt die Absolutdauer der einzuhaltenden Frist von der jeweiligen Anwendung ab: in der Regel ergeben sich Zeitanforderungen an einen Prozess dadurch, dass er –zum Beispiel als Regel- oder Steuereinheit– mit einem externen technischen Prozess interagiert. Die Zeiteigenschaften dieses technischen Prozesses sind entscheidend für die Bemessung der Fristen: während beispielsweise bei einer Temperaturregelung des Innenraums eines Fahrzeugs Fristen in der Größenordnung von Minuten angemessen sind, muss das Auslösen eines Airbags nach einem Aufprall innerhalb weniger Millisekunden erfolgen.

Aufgaben mit Echtzeitanforderungen müssen über gesicherte Mengen an Rechenkapazität verfügen können, d.h. diese Kapazitäten müssen in der Regel vorab bereitgestellt werden, und sie müssen so bemessen sein, dass jede dieser Aufgaben innerhalb ihrer jeweiligen Frist abgeschlossen werden kann. Bei den „harten" Echtzeitanforderungen darf es auch unter den denkbar ungünstigsten Bedingungen nicht zu einer Fristüberschreitung kommen.

In diesem Abschnitt werden zunächst die nötigen Grundbegriffe sowie eine einheitliche Notation zur Beschreibung der Echtzeitverarbeitung eingeführt, auf die im Rest der Arbeit Bezug genommen wird. Dazu wird ein an [ZA95] angelehntes Modell für Echtzeitprozesse eingeführt. Im Anschluss werden verschiedene Verfahren der Prozessplanung und Prozessorzuteilung vorgestellt. Ein Modell zur Beschreibung von Nicht-Echtzeitprozessen wird in 2.2.2 eingeführt.

2.2.1 Echtzeitprozessmodell

Ein Prozess beschreibt ein Programm in Ausführung, d.h. eine Abfolge von Programmschritten, die von einem Prozessor ausgeführt werden. Die Ausführung folgt einem Pfad durch den Programmcode. Bei den meisten praktischen Prozessen ist dieser Ausführungspfad endlich, d.h., es gibt einen Startpunkt, von dem ausgehend die Ausführung zielgerichtet fortschreitet und der Prozess endet, wenn das Ziel erreicht wird (siehe Abbildung 2.4). Daneben gibt es jedoch auch die Möglichkeit nicht-endlicher Prozesse, deren Ausführung ebenfalls einem Pfad durch den Programmcode folgt, bei denen es jedoch weder einen Start- noch einen Zielpunkt gibt. Da die Menge des Programmcodes und damit die Anzahl möglicher Pfade auch für einen nicht-endlichen Prozess endlich ist, muss sein Ausführungspfad letztendlich zirkulär verlaufen (siehe Abbildung 2.4). Ein einfaches Beispiel für einen solchen nicht-endlichen Prozess ist der „idle"-Prozess, den es in jedem Betriebssystem gibt. Sein Programmcode besteht gewöhnlich aus einer leeren Endlosschleife.

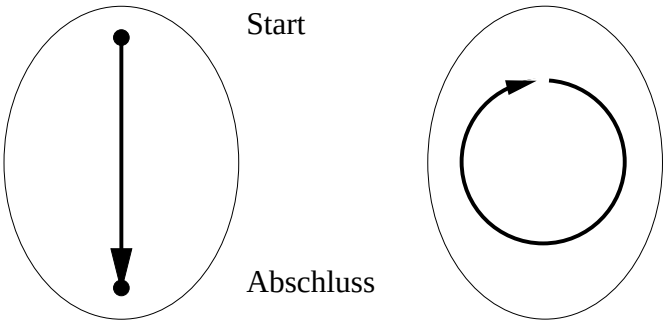

Abbildung 2.4: Prozess als sequenzielle, endliche (links) oder nicht-endliche (rechts) Ausführungsfolge von Anweisungen (vgl: [ZA95], S.24).

In der Echtzeitplanung werden in der Regel ausschließlich endliche Prozesse betrachtet. Daher sind, sofern im Folgenden Prozesse nicht ausdrücklich als „nicht-endlich" bezeichnet werden, stets endliche Prozesse gemeint. Die Aufgabe der Echtzeitplanung besteht darin, solchen Prozessen Betriebsmittel derart zuzuteilen, dass die von der Anwendung vorgegebenen Zeitbedingungen eingehalten werden. Abhängig von den Erfordernissen der Anwendung kann ein endlicher Prozess unterbrechbar (engl: *preemptive*) oder nicht-unterbrechbar (engl: *non-preemptive*) sein:

- Ein *nicht-unterbrechbarer* Prozess kann zwischen Start und Ziel nicht unterbrochen werden.

- Die Ausführung eines *unterbrechbaren* Prozesses kann –sofern keine besonderen Einschränkungen gegeben sind– an beliebiger Stelle unterbrochen werden.

Prinzipiell kann auch ein nicht-endlicher Prozess nicht-unterbrechbar sein, allerdings würde ein solcher Prozess seinen Prozessor vollständig monopolisieren, sodass jegliche Ausführungsplanung müßig wäre. Von den nicht-endlichen Prozessen sind daher nur die Unterbrechbaren praktisch relevant.

Bei einer *Prozessumschaltung* wird ein Prozessor an einen anderen Prozess vergeben. Prozessumschaltungen können –ebenso wie der Start eines Prozesses– entweder durch ein externes Ereignis (z.B. ein Signal von einem zu überwachenden technischen System) oder durch das Rechensystem selbst (z.B. aufgrund des Erreichens einer Zeitmarke) ausgelöst werden. Letzteres geschieht in den meisten Fällen periodisch wiederholt: Ein Prozess, der jeweils nach Ablauf einer festen Zeitspanne neu gestartet wird, ist ein *periodischer* Prozess. Dagegen wird ein Prozess, der -z.B. durch ein externes Ereignis- in unregelmäßigen, nicht vorherbestimmbaren Zeitabständen wiederholt gestartet wird, als *aperiodisch* bezeichnet. Aperiodische Prozesse sind aufgrund ihrer Unbestimmtheit kaum sinnvoll einplanbar. Bei den meisten technischen Systemen lässt sich aber ein minimaler Zeitabstand angeben, der mindestens zwischen zwei aufeinanderfolgenden Starts eines aperiodischen Prozesses liegt. Ist dies der Fall, so wird der betreffende aperiodische Prozess auch als *sporadisch* bezeichnet (vgl. [But97], S.109).

Eine Beschreibung der Aufgabe eines Prozesses (Beispiel: das Abfragen eines Sensorwertes) liegt in Form seines Programmcodes vor. Dies wird als *Prozesstyp* P bezeichnet. Ein solcher Prozesstyp kann mit unterschiedlichen Daten mehrfach instanziiert werden (Beispiel: das Abfragen der Sensorwerte verschiedener, gleichartig aufgebauter Sensoren). Dies wird als die *Ausführung eines Prozessobjektes*, P_i, bezeichnet. Ein Prozessobjekt P_i kann nach Ende einer Ausführung erneut gestartet werden (das gilt insbesondere für periodische Prozesse). Dies wird dann als die j-te Ausführung des Prozessobjektes, P_i^j, bezeichnet (Beispiel: zyklisch wiederholtes Abfragen eines Sensorwertes).

Einem Prozessobjekt P_i werden folgende charakteristische Zeitpunkte und -spannen zugeordnet:

Bereitzeit (engl: *ready time*) r_i:
Der früheste Zeitpunkt, zu dem ein Prozessor an P_i zugeteilt werden kann.

Startzeit (engl: *starting time*) s_i:
Zu diesem Zeitpunkt beginnt der Prozess seine Ausführung durch einen Prozessor.

Ausführungszeit (engl: *execution time*) Δe_i:
Dies ist die zur Prozessausführung benötigte Zeitspanne. Diese Zeit hängt in der Praxis stark von den Daten der Ausführung und dem Zustand des Rechensystems zum Zeitpunkt des Beginns der Ausführung ab. Hier wird die abhängig von diesen Daten längst-mögliche Ausführungszeit (engl: *worst case execution time*) zugrunde gelegt.

Abschlusszeit (engl: *completion time*) c_i:
Zu diesem Zeitpunkt beendet der Prozess seine Ausführung durch einen Prozessor.

Frist (engl: *deadline*) d_i:
Dies ist der Zeitpunkt, zu dem die Prozessausführung beendet sein muss.

Dabei sind Bereitzeit, Ausführungszeit und Frist von der Anwendung vorgegebene Größen, während Start- und Abschlusszeit durch die Echtzeitplanung festzulegen sind. Für nicht-unterbrechbare Prozesse ist die Ausführungszeit Δe_i gleich der Dauer des Zeitintervalls zwischen Start- und Abschlusszeit $[s_i, c_i]$, für unterbrechbare Prozesse ist sie über dieses Intervall verstreut (siehe Abbildung 2.5).

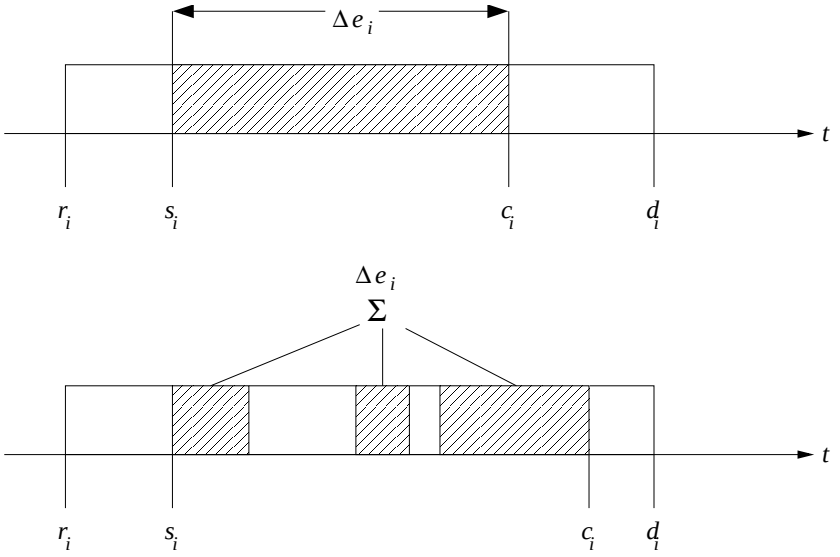

Abbildung 2.5: Zeitdiagramm einer ununterbrochenen (oben) und einer unterbrochenen (unten) Ausführung eines Prozesses P_i auf einem Prozessor.

Für periodische Prozesse eignet sich eine leicht modifizierte Form der Beschreibung: Bereitzeiten und Fristen werden durch die *Periode* bestimmt.

Periode (engl: *period*) Δp_i:
Die Zeitspanne Δp_i definiert für einen periodischen Prozess den Rahmen seiner j-ten Ausführung, P_i^j.

Der Anfangszeitpunkt der j-ten Periode ist zugleich die Bereitzeit r_i^j der j-ten Ausführung P_i^j, der Endzeitpunkt j-ten Periode (und damit zugleich der Anfangszeitpunkt der $(j+1)$-ten Periode) ist die Frist der j-ten Ausführung P_i^j (siehe Abbildung 2.6).

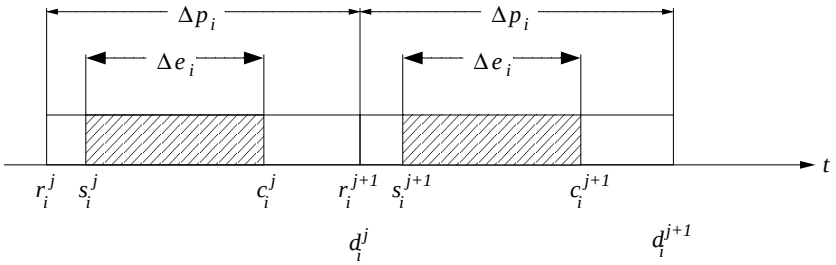

Abbildung 2.6: Die j-te und die $(j+1)$-te ununterbrochene Ausführung von P_i^j und P_i^{j+1}.

Eine weitere charakteristische Größe eines periodischen Prozesses ist seine *Auslastung*:

Auslastung (engl: *utilisation*) U_i:
Beschreibt den Anteil der Periode, währenddessen der Prozess den Prozessor nutzt:

$$U_i = \frac{\Delta e_i}{\Delta p_i} \tag{2.2}$$

Die Auslastung des Systems durch einen periodischen Prozesses gibt den durchschnittlichen Anteil an, den der betreffende Prozess an der gesamten Rechenkapazität des Rechensystems hat. Auch aperiodische Prozesse lassen sich, sofern sie auch sporadisch sind (s.o.), durch eine Auslastung charakterisieren: Wie erwähnt, kann für einen sporadischen Prozess die minimale Zeitspanne zwischen zwei aufeinanderfolgenden Bereitzeiten des Prozesses angegeben werden. Im ungünstigsten Fall, d.h. bei maximaler Wiederholungsrate, wird der sporadische Prozess somit zu einem periodischen Prozess, dessen Periodendauer Δp_i gleich dieser minimalen Zeitspanne ist. Mit seiner Ausführungszeit Δe_i lässt sich somit eine maximale Auslastung durch den aperiodischen Prozess angeben, die bei der Planung zu berücksichtigen ist.

Die Auslastung des Systems durch eine Menge P periodischer Prozesse ist die Summe der Auslastungen durch die einzelnen Prozesse:

$$U(P) = \sum_{i \in P} U_i = \sum_{i \in P} \frac{\Delta e_i}{\Delta p_i} \tag{2.3}$$

Die Auslastung ist ein wichtiges Kriterium, um zu entscheiden, ob eine gegebene Menge von Prozessen auf einem Rechensystem ausgeführt werden kann, ohne dass es dabei zu Fristüberschreitungen kommt. Offensichtlich kann ein System maximal zu 100% ausgelastet werden, d.h. liegt die durch eine Prozessmenge gegebene Auslastung über 1, so kann diese Prozessmenge auf einem Einprozessorsystem mit Sicherheit nicht ohne Fristverletzung ausgeführt werden. Dass die Auslastung unter 100% liegt ist ein notwendiges, aber kein hinreichendes Kriterium für die Planbarkeit. In Abschnitt 2.2.7 werden zwei Planungsverfahren vorgestellt, die auch hinreichende Kriterien für die Planbarkeit einer Prozessmenge liefern.

2.2.2 Nicht-Echtzeitprozesse

Das in 2.2.1 eingeführte Prozessmodell befasst sich mit Prozessen, die Zeitbedingungen unterliegen, und die Echtzeitplanung beschränkt sich auf endliche Prozesse. In den in dieser Arbeit betrachteten Systemen existieren aber darüber hinaus auch Prozesse, die nicht in diese Kategorie passen, da sie keinen Zeitbedingungen unterliegen und da sie auch nicht notwendigerweise endlich sind. Ein einfaches Beispiel für einen solchen nicht-endlichen Prozess ohne Zeitanforderungen ist der „idle"-Prozess, den jedes Betriebssystem enthält: Dieser Prozess hat die einzige Aufgabe, überschüssige Rechenleistung zu konsumieren. Zu diesem Zweck muss er stets rechenbereit sein, um immer dann ausgeführt werden zu können, wenn im System kein anderer rechenbereiter Prozess existiert. Ein solcher, stets rechenbereiter Prozess wird auch als „gierig" (engl.:*greedy*) bezeichnet. Der Programmcode eines „idle"-Prozesses besteht daher traditionell aus einer leeren Endlosschleife. Bei aktuellen Architekturen ist es auch möglich, stattdessen den Prozessor mit Hilfe eines entsprechenden Maschinenbefehls anzuhalten, um den Energieverbrauch des Systems während der inaktiven Phase zu reduzieren. Von außen betrachtet ändert das aber nichts an den Eigenschaften des „idle"-Prozesses, d.h. er kann nach wie wie vor als gieriger, nicht-endlicher Prozess angesehen werden.

Für nicht-endliche Prozesse können offensichtlich weder Bereitzeiten noch Ausführungszeiten oder Fristen angegeben werden. Für endliche Nicht-Echtzeitprozesse gibt es zwar Bereitzeiten und Ausführungszeiten aber ebenfalls keine Fristen. Damit existieren für Nicht-Echtzeitprozesse keine Parameter, nach denen eine

Ausführungsplanung der in 2.2.1 beschriebenen Form vorgehen könnte. Die Prozessorzuteilung erfolgt für solche Prozesse üblicherweise nach einer „best effort"-Methode (z.B. dem Zeitscheibenverfahren, siehe 2.2.8), d.h. alle verfügbare Rechenzeit wird vollständig und möglichst gleichmäßig über die vorhandenen Nicht-Echtzeitprozesse verteilt. Mitunter kommen auch heuristische Methoden zum Einsatz, beispielsweise, um durch eine optimierte Nutzung des Cache den Gesamtdurchsatz des Systems zu verbessern oder um durch Bevorzugung interaktiv arbeitender Prozesse das System aus Anwendersicht schneller erscheinen zu lassen.

2.2.3 Parallelprozesse

Das bisher beschriebene Prozessmodell entspricht der Arbeitsweise eines Automaten: Einzelne Anweisungen werden nacheinander ausführt, wobei das ausgeführte Programm Funktion und Reihenfolge der Anweisungen vorgibt. Tatsächlich besteht die Tätigkeit des „Programmierens" in einer imperativen Programmiersprache im Wesentlichen darin, eine erwünschte Funktionalität in eine Folge von Arbeitsschritten zu übersetzen. Dabei werden zeitliche Abfolgen der Arbeitsschritte vorgegeben, auch wenn dies für das Erbringen der geforderten Funktion nicht erforderlich wäre.

```
sum1   = a + b;
sum2   = c + d;
result = sum1 *  sum2;
```

Listing 2.1: *Beispiel: Berechnung von* $(a + b) \cdot (c + d)$

Als Beispiel zeigt Listing 2.1, wie etwa die Berechnung des arithmetischen Ausdruckes $(a + b) \cdot (c + d)$ in der Programmiersprache „C" aussehen könnte. Dabei ist die Reihenfolge der Berechnung der Zwischensummen `sum1` und `sum2` vom Programmierer willkürlich festgelegt worden, sie hätte ebensogut auch vertauscht werden können. Für eine korrekte Funktion ist nur wichtig, dass beide Zwischensummen dann vorliegen, wenn das Resultat `result` berechnet wird. Durch die imperative Programmierung wird hier ein Freiheitsgrad aufgegeben. Bei einer rein funktionalen Sprache wäre dies nicht erforderlich. Dies sollte allerdings nicht darüber hinwegtäuschen, dass nicht die verwendete Programmiersprache, sondern letztendlich die Eigenschaften der Maschine selbst verantwortlich sind: Compiler und Laufzeitumgebungen für funktionale Sprachen bilden das funktionale Programmiermodell auf das sequenzielle Modell des Prozessors ab, d.h. letzten Endes erzeugen auch sie wieder sequenziell auszuführende Programme.

Stehen in obigem Beispiel mehrere Prozessoren zur Verfügung, so kann jeder von ihnen eigenständig einen Programmpfad beschreiten. Damit wäre es möglich, die

beiden Zwischensummen gleichzeitig, mit verschiedenen Prozessoren zu berechnen. Die Berechnung des Resultates kann allerdings auch dann erst beginnen, wenn die Zwischensummen vorliegen.

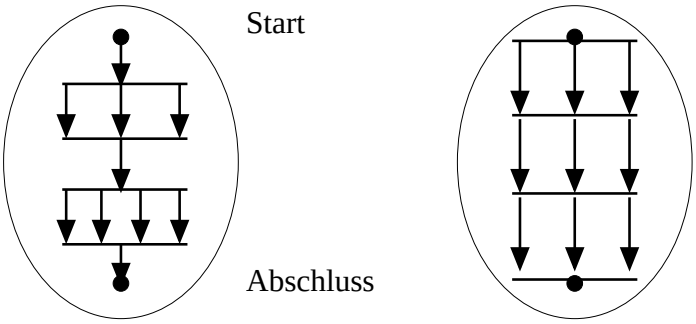

Abbildung 2.7: Parallelprozesse.

An diesem Beispiel werden die grundlegenden Eigenschaften eines Parallelprozesses erkennbar (siehe Abbildung 2.7): er besteht aus mehreren Sequenzen von Programmschritten, also sequenziellen Teilprozessen, die zumindest teilweise unabhängig voneinander sind. Diese Teilprozesse folgen (potenziell) verschiedenen Pfaden. Falls ein Teilprozess von den Ergebnissen eines oder mehrerer anderer Teilprozesse abhängt (im Beispiel trifft das auf die Berechnung des Resultats zu), so muss eine Synchronisierung erfolgen: der abhängige Teilprozess kann erst beginnen, wenn die Ergebnisse aller Teilprozesse, von denen er abhängt, vorliegen. Dazu gibt es Synchronisationspunkte, an denen Teilprozesse enden und neue beginnen. Häufig trennen solche Synchronisationspunkte die einfädigen (sequenziellen) und mehrfädigen (parallelen) Anteile der Verarbeitung voneinander, wie in der linken Hälfte von Abbildung 2.7 angedeutet. Allgemein aber können an einem Synchronisationspunkt beliebig viele Teilprozesse enden und beliebig viele neue entspringen (s. Abbildung 2.7, rechts). Bei einem endlichen Parallelprozess sind insbesondere auch die Anfangs- und Endpunkte Synchronisationspunkte.

Die Teilprozesse können mit den bereits eingeführten Mitteln beschrieben werden: Da sie in der Regel unterschiedliche Anweisungsfolgen ausführen, handelt es sich um Prozesstypen, deren Prozessobjekte periodisch oder sporadisch ausgeführt werden. Diesen Prozessobjekten können die in 2.2.1 vorgestellten charakteristischen Zeitpunkte und -spannen zugeordnet werden. Allerdings ergeben sich Unterschiede bezüglich der Definition der Auslastung: Die Gleichung (2.3) gibt die Auslastung eines Prozessors an. Die Auslastung eines Systems mit m Prozessoren muss auf diese Prozessorzahl bezogen werden:

$$U_{system} = \frac{1}{m} \cdot \sum_{i \in P} \frac{\Delta e_i}{\Delta p_i} \tag{2.4}$$

Die Summe der Auslastungen U_i durch eine Menge periodischer Prozesse nach Gleichung (2.2) kann damit größer als 1 (maximal gleich m) werden, bevor eine Planbarkeit mit Sicherheit ausgeschlossen ist. Die Auslastung des Systems U_{system} kann hingegen nach wie vor maximal gleich 1 sein.

2.2.4 Parallelverarbeitung

Wie in Abschnitt 2.2.3 dargestellt, können Teilprozesse eines Parallelprozesses aufgrund ihrer gegenseitigen Unabhängigkeit in beliebiger Reihenfolge ausgeführt werden. Stehen mehrere physische Prozessoren zur Verfügung, so kann daher jeder Teilprozess auf einem eigenen Prozessor ausgeführt werden, wobei sich die schnellstmögliche Abarbeitung dann ergibt, wenn alle Teilprozesse gleichzeitig auf ihren verschiedenen Prozessoren arbeiten. Dazu muss die Anzahl der physischen Prozessoren größer oder gleich der Anzahl der Teilprozesse sein. Die Idealsituation einer maximalen Auslastung der Prozessoren ergibt sich, wenn die Anzahl der Prozessoren gleich der Anzahl der Teilprozesse ist.

Ein speziell für die parallele Abarbeitung konstruiertes Programm kann ggf. statisch oder sogar dynamisch auf die jeweils verfügbare Prozessoranzahl abgestimmt werden. Im Umfeld der Konsolidierung eingebetteter Systeme ist davon auszugehen, dass die abzuarbeitende Rechenlast zumindest teilweise aus solchen angepassten Parallelprozessen besteht. Daneben muss aber auch mit Parallelprozessen gerechnet werden, deren Teilprozessanzahl unter oder über der der Prozessoren liegt. Im ersten Fall bleibt ein Teil der Prozessoren ungenutzt, im zweiten Fall müssen zumindest einige der Teilprozesse nacheinander auf einem gemeinsamen Prozessor ausgeführt werden.

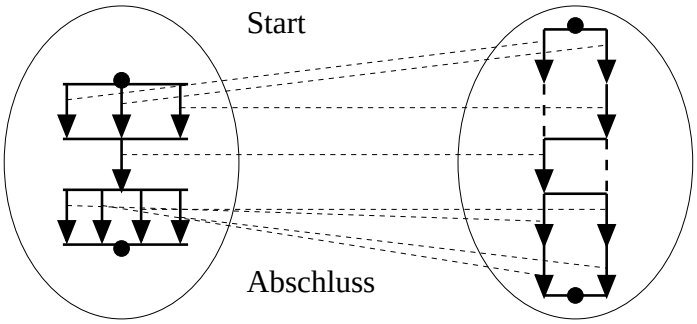

Abbildung 2.8: Teil-sequenzielle Ausführung eines Parallelprozesses auf zwei Prozessoren.

Abbildung 2.8 zeigt, wie ein Parallelprozess bei der Ausführung auf eine Zweiprozessorsystem teilweise sequenziell ausgeführt werden muss. Dabei stellen die

gestrichelt dargestellten Teile des Pfades Zeitbereiche dar, während derer einer der Prozessoren ungenutzt bleibt.

Solange nur die funktionalen Eigenschaften der Programme betrachtet werden, ist es gleichgültig, ob die Teilprozesse sequenziell oder parallel ausgeführt werden. Rechensysteme, die keine Echtzeitanforderungen zu erfüllen haben, können daher alle Prozesse in gleicher Weise behandeln: Gleichgültig, ob es sich um Teilprozesse eines parallelen Prozesses oder um einzelne, unabhängige Prozesse handelt, die Prozessorzuteilung kann einfach nach der Maxime der besten Auslastung erfolgen, da sich so auch der größtmögliche Gesamtfortschritt über alle Prozesse ergibt.

Geht es jedoch neben der funktionalen Korrektheit noch um eine möglichst schnelle Bearbeitung einer Aufgabe, so ergibt sich diese bei einer gleichmäßigen Verteilung der zu dieser Aufgabe gehörenden Rechenlast auf alle verfügbaren Prozessoren, wobei die kleinste Einheit der Verteilung durch den einzelnen Teilprozess gegeben ist. Daher müssen Echtzeitsysteme zusätzlich berücksichtigen, dass für alle Teilprozesse eines parallelen Prozesses eine gemeinsame Bereitzeit und Frist gilt. Da die Fähigkeit, Fristen einhalten zu können, die entscheidende Eigenschaft eines Echtzeitsystems darstellt, kommt dem eingesetzten Prozessor-Zuteilungsverfahren hier eine besondere Bedeutung zu.

Coscheduling

Ein bei der Parallelverarbeitung häufig auftretender Sonderfall sind Gruppen von Teilprozessen, die jeweils gemeinsam aus einem Synchronisierungspunkt entstehen und gemeinsam in einem anderen Synchronisierungspunkt enden (siehe Abbildung 2.7, rechte Seite). Alle Teilprozesse einer solchen Gruppe haben gemeinsame Bereitzeiten und Fristen. Derartige Teilprozessmengen ergeben sich immer dann, wenn eine Aufgabe „parallelisierbar" ist, d.h. wenn sie sich gut in voneinander unabhängige Teilaufgaben aufspalten lässt. Die Aufgabe ist erst dann erfüllt, wenn der letzte der Teilprozesse das Ziel erreicht hat, d.h. die längste aller Ausführungszeiten bestimmt die Frist. Um eine möglichst kurze Frist einhalten zu können, muss einerseits die zu erbringende Arbeit möglichst gleichmäßig auf alle zur Verfügung stehenden Prozessoren verteilt werden, und andererseits müssen die Teilprozesse bei der Planung als eine Einheit behandelt werden, sodass alle stets gleichzeitig arbeiten. In [Ous80] bezeichnet Ousterhout eine solche Teilprozessmenge, die auf ein gemeinsames Ziel hinarbeitet, als „Task Force". Für die Vorgehensweise, alle Teilprozesse einer Task Force stets gemeinsam auszuführen, wurde der Begriff „coscheduling" geprägt. Ousterhout zeigt in seiner Arbeit, dass es für Teilprozesse einer Task Force u.U. vorteilhaft sein kann, die Synchronisie-

rung durch *aktives* Warten anstatt wie sonst üblich durch blockierende Synchronisationsprimitive zu bewerkstelligen: Da ohnehin jeder der Teilprozesse parallel auf einem eigenen Prozessor arbeitet, bringt das Blockieren (d.h. das Freigeben des eigenen Prozessors) keine Vorteile für die anderen Mitglieder der Task Force. Zudem ist das Blockieren mit Verlusten durch Prozesswechsel verbunden. Da alle Mitglieder in etwa das gleiche Arbeitspensum haben, sollten auch alle in etwa zur gleichen Zeit den Synchronisationspunkt erreichen, sodass das aktive Warten in der Regel schnell beendet sein wird. Die Prozesswechselverluste, die bei einer Synchronisation mit blockierendem Warten anfallen würden, übersteigen die durch dieses kurze aktive Warten entstehenden Verluste erheblich, sodass hier aktives Warten nicht nur aus Geschwindigkeits- sondern auch aus Effizienzgründen vorteilhaft ist.

Granularität

Je nach der Häufigkeit von Synchronisationen zwischen den Programmpfaden spricht man von grob- bzw. mittelgranularer[1] Parallelverarbeitung (vgl. [Sta05], S. 442). Sind die Programmpfade in weiten Teilen unabhängig voneinander, d.h. handelt es sich um eine grobgranulare Parallelverarbeitung, so können sie als eigenständige, endliche oder nicht-endliche, sequenzielle Prozesse aufgefasst werden. Die (wenigen) vorhandenen Abhängigkeiten können durch die Programmierung explizit erfüllt werden, beispielsweise mit Hilfe der üblichen Mittel zur Interprozesskommunikation.

Für die mittelgranulare Parallelverarbeitung ist diese Vorgehensweise jedoch unhandlich. Zur Implementierung solcher Prozesse gibt es spezielle Konzepte wie CSP ([Hoa78]) und darauf basierende, spezielle Programmiersprachen wie z.B. Occam ([MT84]). Auch für klassische, sequenzielle Programmiersprachen gibt es Erweiterungen zur parallelen Programmierung, wie z.B. OpenMP [CJP07] (wobei hier im Gegensatz zu CSP die Kommunikation nicht über Nachrichten, sondern über gemeinsamen Speicher erfolgt).

Auch in diesen Sprachen stellt sich ein Parallelprozess als eine Menge von Teilprozessen dar, die teilweise voneinander abhängig sind. Die Sprache macht hier entweder direkte Vorgaben (zum Beispiel wird bei Occam jede einzelne Anweisung als Teilprozess aufgefasst), oder es liegt in der Verantwortung des Programmierers, die Grenzen der Teilprozesse explizit festzulegen. Listing 2.2 zeigt als Beispiel hierfür ein C-Codefragment, bei dem eine `for`-Schleife mit Hilfe eines

[1] **Fein**granulare Parallelverarbeitung bezeichnet laut [Sta05] die Parallelverarbeitung auf der Ebene der Maschineninstruktionen, wie sie z.B. bei „Superskalar-"Prozessoren zu finden ist [PH05]. Diese ist aus Programmierersicht transparent (vergl. 2.1.2).

`#pragma`-Konstrukts als „parallelisierbar" gekennzeichnet ist. Dieses Konstrukt wird von einem –in diesem Beispiel OpenMP unterstützenden– Compiler erkannt, der daraus mehrfädigen Code generiert.

```
#pragma omp parallel for
    for (i = 0; i < numPixels; i++)
{
        pGrayScaleBitmap[i] = (unsigned BYTE)
        (pRGBBitmap[i].red   * 0.299 +
         pRGBBitmap[i].green * 0.587 +
         pRGBBitmap[i].blue  * 0.115);
}
```

Listing 2.2: *Beispiel: Parallele Programmierung in C (Quelle: [AR06], S.136).*

Grobe und mittelgranulare Parallelverarbeitung unterscheiden sich nicht grundsätzlich bezüglich ihres Prozessmodells: Beide lassen sich als Menge unabhängiger, parallel arbeitender Teilprozesse und dazwischen liegende Synchronisationspunkte abstrahieren. Zur Umsetzung der Synchronisationspunkte werden allerdings unterschiedliche Mittel eingesetzt.

2.2.5 Echtzeitplanung

Für die Echtzeitplanung sind Bereitzeiten, Ausführungszeiten, Fristen, sowie ggf. Perioden und Auslastungen als durch die Anwendung vorgegebene Eingabeparameter zu betrachten. Die Aufgabe besteht dann darin, Start- und Abschlusszeiten, sowie bei unterbrechbaren Prozessen zusätzlich die Folge von Unterbrechungsintervallen derart festzulegen, dass es zu keinerlei Fristüberschreitung kommt. Sofern dabei alle Daten vorab bekannt sind, kann auch vorab ein entsprechender Plan erstellt werden, der dann zur Laufzeit ausgeführt werden muss (*statische* Echtzeitplanung). Fallen die Daten hingegen erst im Laufe der Prozessausführung an, so muss auch die Echtzeitplanung *dynamisch*, d.h. parallel zur Prozessausführung erfolgen. Die Ergebnisse der Echtzeitplanung können entweder in Form eines vollständigen Ausführungsplanes (explizite Planung), oder in Form von Regeln (implizite Planung) vorliegen.

Die Echtzeitplanung gliedert sich in drei wesentliche Phasen:

1. Einplanbarkeitstest (engl.: *feasibility check*): Hierbei wird geprüft, ob für die gegebene Menge von Prozessen überhaupt ein Plan ermittelt werden kann, der eine fristgerechte Ausführung aller Prozesse garantiert.

2. Planung (engl.: *scheduling*): Während der Einplanbarkeitstest nur die Aussage der Möglichkeit einer fristgerechten Planung liefert, wird in dieser Phase der eigentliche Plan erstellt. Dieser Plan liefert eine eindeutige Zuordnung von Prozessen zu Prozessoren.

3. Prozessorzuteilung (engl.: *dispatching*): Durch entsprechende Prozessumschaltungen wird der Plan ausgeführt.

Bei der statischen Echtzeitplanung werden sowohl der Einplanbarkeitstest als auch die Planung vorweg, also während der Systemkonfiguration vorgenommen. Lediglich die Prozessorzuteilung geschieht dann zur Laufzeit des Systems. Bei der dynamischen Echtzeitplanung hingegen werden alle drei Phasen zur Laufzeit ausgeführt.

2.2.6 Zeit- und Ereignissteuerung

Eine etwas andere Sichtweise der Unterscheidung zwischen statischer und dynamischer Planung findet sich bei Kopetz: Nach [Kop91] kann die Ablaufsteuerung in Echtzeitsystemen nach zwei grundverschiedenen Prinzipien erfolgen:

- *Zeitsteuerung*: Hierbei liegt ein vorab erstellter Zeitplan vor, der die Bereitzeiten der Prozesse als Funktion der Zeit angibt. In der Regel wird dieser Zeitplan zyklisch wiederholt.

- *Ereignissteuerung*: Hierbei wird das System durch eine nicht vorher absehbare Folge von Ereignissen getrieben: Prozesse werden durch externe Ereignisse aktiviert.

Die Zeitsteuerung entspricht im Wesentlichen der zuvor eingeführten statischen Planung, während die Ereignissteuerung der dynamischen Planung entspricht. Allerdings kann hier die Art der Steuerung als Eigenschaft des jeweiligen Prozesses, nicht als eine globale, alle Prozesse betreffende Eigenschaft des Systems angesehen werden, d.h. ein einzelner Prozess ist so programmiert, dass er entweder zeit- oder ereignisgesteuert arbeitet, je nachdem, welche Variante den durch das externe physikalisch/technische System definierten Anforderungen am besten entspricht.

Sowohl bei zeit- als auch bei ereignisgesteuerten Prozessen wird die Qualität des Ergebnisses anhand der Länge der einhaltbaren Fristen und Latenzzeiten sowie deren Variation (dem „Jitter") beurteilt. Bei zeitgesteuerten Echtzeitprozessen sind dabei Fristen und Latenzzeiten jeweils im Bezug auf absolute Startzeiten

definiert, bei ereignisgesteuerten Prozessen werden sie auf den Zeitpunkt des auslösenden Ereignisses bezogen. Bei beiden Konzepten können sowohl „harte" als auch „weiche" Echtzeitanforderungen gelten.

In der Praxis sind rein ereignis- oder rein zeitgesteuerte Systeme eher selten. In nahezu allen Fällen gibt es zugleich Anwendungsteile, die dem einen oder dem anderen Ansatz entsprechen [Kai06]. Insbesondere eine Plattform zur Konsolidierung eingebetteter Systeme muss beides unterstützen, da eine solche Plattform gleichzeitig eine Vielzahl verschiedener Echtzeit-Anwendungen tragen können muss. Die Koexistenz von zeit- und ereignisgesteuerten Prozessen in einem System führt allerdings zu Konflikten, da die beiden Varianten konträr und nur unter Abstrichen miteinander vereinbar sind: Erlaubt man beispielsweise, dass ereignisgesteuerte Prozesse mit dem Ziel einer schnellen Reaktion sofort auf Ereignisse ansprechen können, so unterbrechen sie dadurch möglicherweise den vorgegebenen Ablauf zeitgesteuerter Prozesse und verschlechtern damit deren Zeitverhalten. Erzwingt man umgekehrt die Unterordnung ereignisgesteuerter Prozesse unter den starren Ablaufplan der Zeitsteuerung, etwa indem man ihnen feste Zeitscheiben innerhalb dieses Ablaufplans zuweist (vgl. [But97], S.111), so werden Reaktionen auf externe Ereignisse immer bis zum Eintritt der nächsten Zeitscheibe verzögert, d.h. in diesem Fall leidet das Zeitverhalten der ereignisgesteuerten Prozesse.

Immer, wenn zeit- und ereignisgesteuerte Prozesse in einem System aufeinandertreffen, muss einem der beiden Konzepte Vorrang eingeräumt werden, und als direkte Konsequenz dieser Entscheidung muss in Kauf genommen werden, dass die zeitliche Determiniertheit der jeweils anderen Seite dadurch verschlechtert wird. Dieses Dilemma ist –zumindest auf Einprozessormaschinen– grundsätzlich nicht lösbar ([Foh99]). Bei Mehrprozessormaschinen kann hingegen eine Entkopplung der beiden Konzepte erreicht werden, indem zeit- und ereignisgesteuerte Prozesse an verschiedene physische Prozessoren gebunden werden.

2.2.7 Planungsverfahren

Die Probleme des Einplanbarkeitstests, also der Entscheidung, ob für eine gegebene Menge von Prozessen ein brauchbarer Plan existiert, und ggf. der Planerstellung, also des systematischen Auffindens eines solchen Plans, sind vielfach studiert worden. Eine grundlegende Arbeit, deren Aussagen und Ergebnisse bis heute Bestand haben, wurde 1973 von Liu und Layland vorgelegt [LL73].

In der genannten Arbeit werden Mengen unterbrechbarer periodischer Prozesse betrachtet, die voneinander unabhängig sind. Jeder Prozess i wird durch seine individuelle Periodendauer Δp_i und seine Ausführungszeit Δe_i charakterisiert. Der

Beginn einer Periode ist dabei zugleich die Bereitzeit für die jeweilige Ausführung des Prozesses und das Ende der Periode gilt gleichzeitig als Frist. Liu und Layland beschreiben zwei verschiedene Planungsverfahren:

1. Das *Planen nach monotonen Raten* (engl.: *Rate monotonic scheduling – RMS*) gibt eine Zuordnung von Prioritäten zu den Prozessen vor, die zur Laufzeit nicht verändert wird. Man spricht deshalb bei RMS auch vom „Planen mit statischen Prioritäten" (engl.: *static priority scheduling*). Ausgeführt wird dann jeweils derjenige rechenbereite Prozess, der über die derzeit höchste Priorität verfügt.

2. Beim *Planen nach Fristen* (engl.: *Earliest deadline first – EDF*) wird stets derjenige rechenbereite Prozess ausgeführt, dessen Frist am kürzesten ist. Falls die Prozesssteuerung des zugrundeliegenden Betriebssystems mit Prioritäten arbeitet, so impliziert EDF, dass die den Prozessen zugeordneten Prioritäten –im Gegensatz zum RMS-Verfahren– dynamisch verändert werden. Man spricht deshalb bei EDF auch vom „Planen mit dynamischen Prioritäten" (engl.: *dynamic priority scheduling*). Die Zuordnung bzw. Änderung der Priorität eines Prozesses geschieht bei EDF ausschließlich zu dessen Bereitzeiten und bleibt für die Dauer der Ausführung des Prozessobjektes konstant. Zur Abgrenzung gegenüber anderen Planungsverfahren, die diese Einschränkung nicht haben, bezeichnen wir in dieser Arbeit diese Klasse von Planungsverfahren auch genauer als „auftragsbezogen dynamisch" (engl.: *job-level dynamic*, siehe [CFH+03]).

Beim Planen nach monotonen Raten werden die Prioritäten der Prozesse einfach umgekehrt proportional zu ihren Periodendauern vergeben. Formal ausgedrückt: Für eine Menge unterbrechbarer Prozesse $P = \{1, ..., n\}$ mit den Periodendauern Δp_i, $i \in \{1...n\}$, werden Prioritäten so festgelegt, dass für alle i und j, $i, j \in \{1...n\}$ gilt:

$$i < j \iff prio(i) < prio(j) \tag{2.5}$$

Unter dieser Voraussetzung zeigen Liu und Layland, dass alle Prozesse ihre Fristen einhalten, wenn für die Gesamtauslastung U durch diese Prozessmenge (siehe Gleichung (2.3)) gilt:

$$U \leq n \cdot \left(2^{\frac{1}{n}} - 1\right) \tag{2.6}$$

Für große Anzahlen n von Prozessen wird:

$$\lim_{n\to\infty} n \cdot \left(2^{\frac{1}{n}} - 1\right) = \ln(2) \approx 0,67 \tag{2.7}$$

Liu und Layland zeigen, dass das RMS-Verfahren optimal ist: Sofern überhaupt eine feste Prioritätenzuordnung existiert, die zu einem brauchbaren Plan führt, dann erzeugt auch das RMS-Verfahren einen brauchbaren Plan.

Für das Planen nach Fristen zeigen Liu und Layland in ihrer Arbeit, dass ein brauchbarer Plan immer dann existiert und gefunden wird, wenn für die Gesamtauslastung U durch die Prozessmenge gilt:

$$U \leq 1 \tag{2.8}$$

Da (auf einem Einprozessorrechner) eine Auslastung über 1 nicht möglich ist, bedeutet dies, dass, sofern ein Plan überhaupt existiert, das EDF-Verfahren einen brauchbaren Plan findet.

Beide Planungsverfahren können prinzipiell sowohl bei statischer als auch bei dynamischer Planung eingesetzt werden. Insbesondere das RMS-Verfahren lässt sich, da es mit statischen Prioritäten arbeitet, sehr gut auf die von den meisten Echtzeitbetriebssystemen angebotenen, prioritätsgesteuerten Scheduler-Infrastrukturen abbilden.

2.2.8 Verfahren zur Prozessorzuteilung

Die dritte Phase der Echtzeitplanung, die Prozessorzuteilung, findet immer zur Laufzeit statt. Es gibt im Bereich der Echtzeitsysteme einige unterschiedliche Verfahren zur Prozessorzuteilung, die im Folgenden in Anlehnung an [But97] genauer beschrieben werden.

Der Algorithmus, der in einem System die Zuteilung von Prozessen zu Prozessoren vornimmt, wird, ebenso wie seine Implementierung, in der Literatur häufig einfach als *Scheduler* bezeichnet. Auch hier wird im Folgenden diese Bezeichnung verwendet, wenngleich nach dem in 2.2.5 Gesagten eigentlich die Bezeichnung „Dispatcher" zutreffender erscheint: Die Planerstellung (d.h. das eigentliche „Scheduling") kann, insbesondere bei statischer Planung, durchaus vorab während der Systemkonfiguration stattfinden, d.h. sie ist nicht zwingend Bestandteil des Schedulers. Bei vielen Implementierungen dynamischer Planung hingegen sind die drei Phasen häufig in Form eines gemeinsamen Programms realisiert und dabei so sehr miteinander verwoben, dass ihre Trennung schwer fällt. (Vermutlich liegt darin der Ursprung der Bezeichnung.)

Zeitgesteuerte Prozessorzuteilung

Sind alle nötigen Eingabeparameter für die Echtzeitplanung bekannt, so kann vorab ein vollständiger Plan ermittelt werden. Dieser Plan gibt den jeweils auszuführenden Prozess als Funktion der Zeit an, formal ausgedrückt (vgl. [But97], S.24):

Für eine Prozessmenge $\{P_1, P_2, ...P_n\}$ kann ein Zeitplan als Funktion $\sigma : \mathbb{R}^+ \times \mathbb{N}^0 \rightarrow \mathbb{N}^0$ angegeben werden, so dass $\forall t \in \mathbb{R}^+$ gilt: $\sigma(t) \in \{0, 1, 2, ...n\}$, wobei $\sigma(t) = k > 0$ bedeutet, dass Prozess P_k zur Zeit t arbeitet, und $\sigma(t) = 0$ bedeutet, dass kein Prozess arbeitet. (Dass „kein Prozess arbeitet" bedeutet in der Praxis der Betriebssysteme in der Regel, dass der „idle-"Prozess des Betriebssystems arbeitet).

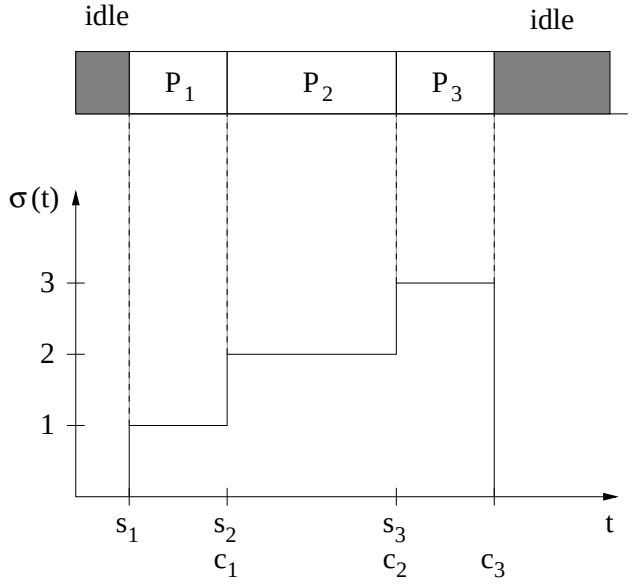

Abbildung 2.9: Ausführungsplan als Funktion der Zeit.

Der Ausführungsplan lässt sich als Treppenfunktion darstellen (siehe Abbildung 2.9). Dieser Plan kann nun rein zeitgesteuert abgearbeitet werden, indem der Scheduler, beispielsweise mit Hilfe eine Hardware-Uhr, jeweils an den Sprungstellen der Funktion $\sigma(t)$ aktiviert wird, um dann zwischen den betreffenden Prozessen umzuschalten. Falls ein Prozess seine Frist überschreitet, falls er also zum vorgesehenen Umschaltzeitpunkt noch nicht zum Abschluss gekommen ist, wird er in jedem Fall unterbrochen. Für andere Prozesse bleibt dieser Vorgang unsichtbar, sodass die Auswirkungen einer Fristverletzung auf denjenigen Prozess beschränkt bleiben, der sie verursacht hat.

Die zeitgesteuerte Prozessorzuteilung ermöglicht eine exakte deterministische Prozessausführung und kann zudem mit sehr geringen Laufzeitverlusten realisiert

werden. Allerdings ist die Voraussetzung der vollständigen Vorab-Planbarkeit oft nicht zu erfüllen: Bei nahezu allen praktischen Anwendungen gibt es Prozesse, deren Planungsdaten erst zur Laufzeit anfallen. Dies trifft zum Beispiel auf alle aperiodischen Prozesse zu. Solche Prozesse können in einer Vorab-Planung allenfalls durch die Annahme einer entsprechenden periodischen Rechenlast berücksichtigt werden (vgl. 2.2.1). Des Weiteren muss der Plan, damit er dem System übergeben werden kann, durch eine endliche Menge an Daten darstellbar sein. Sofern die Laufzeit des Systems unbeschränkt ist, muss er daher entweder mit Hilfe einer geschlossenen Formel berechnet werden können, oder er muss periodisch sein, sodass er nur durch die Angabe der Periodendauer und der Zeitaufteilung innerhalb einer Periode vollständig beschreibbar ist.

Zeitgesteuerte Zuteilung bei Mehrprozessorsystemen

Für eine Anwendung auf Mehrprozessorrechnern muss die oben eingeführte formale Darstellung der zeitgesteuerten Prozessorzuteilung als Zeitfunktion $\sigma(t)$ erweitert werden: Hier kann einem Prozess zu einem gegebenen Zeitpunkt einer von mehreren Prozessoren zugeordnet sein, d.h. der Prozessor kommt als eine weitere Dimension hinzu.

Wir beschreiben einen Zeitplan für eine Prozessmenge $\{P_1, P_2, ...P_n\}$ formal als Funktion $\sigma : \mathbb{R}^+, \mathbb{N}^0, \rightarrow \mathbb{N}^0$, so dass $\forall t \in \mathbb{R}^+, c \in \mathbb{N}^0$ gilt: $\sigma(t, c) \in \{0, 1, 2, ...n\}$, wobei $\sigma(t, c) = k > 0$ bedeutet, dass Prozess P_k zur Zeit t auf Prozessor c arbeitet, und $\sigma(t, c) = 0$ bedeutet, dass zur Zeit t kein Prozess auf Prozessor c arbeitet.

Eine technische Umsetzung einer solchen Prozessorzuteilung wäre dementsprechend ähnlich wie bei mehreren, voneinander getrennt arbeitenden Einprozessorsystemen möglich, d.h. jedem Prozessor wäre eine eigene Hardware-Uhr zugeordnet, die an den Sprungstellen des Zeitplanes den Scheduler aktiviert, damit dieser anhand seines Teils des Plans den jeweiligen Prozessor neu zuteilt. Jeder Teilplan bestünde dabei aus einer Folge zu aktivierender Prozesse und den zugehörigen Aktivierungszeitpunkten. Da hier allerdings für alle Prozessoren eine gemeinsame Zeitkoordinate gilt, müssten die Hardware-Uhren der einzelnen Prozessoren exakt synchronisiert sein. Mit Hilfe geeigneter Protokolle (zum Beispiel dem „Precision Time Protocol" [IEE02]) ist es heute möglich, eine hinreichend genaue Synchronisation der Uhren von Rechnern innerhalb eines verteilten Systems zu realisieren ([Bom08]), womit ein solcher Mehrprozessorzeitplan auch auf NoRMA-Systemen (vgl. 2.1.1) umsetzbar ist.

Im Rahmen dieser Arbeit werden jedoch in erster Linie SMP-Systeme betrachtet, bei denen in der Regel eine allen Prozessoren gemeinsame Hardware-Uhr zum Einsatz kommt. Diese kann entweder einen bestimmten Prozessor oder ei-

ne konfigurierbare Gruppe von Prozessoren unterbrechen. Falls die Möglichkeit besteht, für jede Unterbrechung die Menge der zu unterbrechenden Prozessoren (z.B. durch eine Bitmaske) neu festzulegen, so kann ein solcher Zeitplan, beschrieben als Folge von Prozessen, Prozessormengen und Zeitpunkten, direkt umgesetzt werden: Mit jedem Scheduleraufruf werden nur auf den in der Menge enthaltenen Prozessoren Prozesse umgeschaltet, die Bitmaske wird gesetzt, sodass bei der nächsten Unterbrechung wieder nur diejenigen Prozessoren unterbrochen werden, bei denen ein Prozesswechsel ansteht. Steht eine derart hochentwickelte Hardware-Uhr nicht zur Verfügung, so kann ein analoges Verhalten –allerdings unter Inkaufnahme erhöhter Verluste– auch mit einer fest an einen bestimmten Prozessor gebundenen Uhr erreicht werden. Dazu muss der durch die Uhr unterbrochene Prozessor die übrigen umzuschaltenden Prozessoren mit Hilfe von „Inter-Prozessor-Interrupts" (IPIs) benachrichtigen.

FIFO- und Zeitscheiben-Prozessorzuteilung

Die Prozessorzuteilung nach dem FIFO-Prinzip (engl.: *first in, first out*) ist ein einfaches *dynamisches* Planungs- und Zuteilungsverfahren. Dabei werden Prozesse, sowie sie bereit werden, in eine Warteschlange eingereiht. Der Scheduler teilt den Prozessor jeweils dem ältesten Prozess aus der Warteschlange zu. Dieser Prozess arbeitet daraufhin bis zu seinem Abschluss. Erst dann wählt der Scheduler den nächsten Prozess aus der Warteschlange.

Diese auch als „FCFS" (engl.: *first come, first served*) bezeichnete Methode arbeitet im Gegensatz zur zeitgesteuerten Prozessorzuteilung ausschließlich auf Basis zur Laufzeit anfallender Planungsdaten (den Bereitzeiten der Prozesse). Die Laufzeitverluste dieses Verfahrens sind ähnlich gering wie bei der zeitgesteuerten Prozessorzuteilung, aber es können keine Zeitgarantien gegenüber einzelnen Prozessen abgegeben werden: Da in Ausführung befindliche Prozesse niemals unterbrochen werden, kann ein lang arbeitender Prozess den Start anderer, erst nach ihm bereit gewordener Prozesse sehr lange verzögern. Dadurch ist das Zeitverhalten jedes einzelnen Prozesses vom Verhalten anderer Prozesse abhängig.

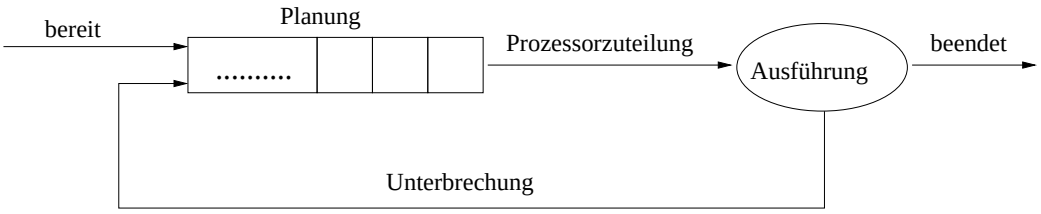

Abbildung 2.10: Schematischer Ablauf der Zeitscheiben-Prozessorzuteilung.

Um diese Abhängigkeit zu verringern, legt das *Zeitscheiben-* oder auch RR-Verfahren (engl.:*Round-Robin*) zusätzlich die Dauer einer *Zeitscheibe* fest. Ist ein Prozess im Besitz des Prozessors, so wird er nach Ablauf dieser Zeitscheibe unterbrochen und an das Ende der Warteschlange gestellt (siehe Abbildung 2.10). Dadurch können andere Prozesse, die später bereit wurden, ebenfalls –maximal für die Dauer einer Zeitscheibe– ausgeführt werden, bevor der unterbrochene Prozess wieder fortgesetzt wird. Die Zeitscheibendauer kann dabei, je nach Anforderung, eine prozessspezifische oder eine globale Größe sein. Auf diese Weise lässt sich eine Verringerung der mittleren Wartezeit und eine gleichmäßigere Aufteilung der verfügbaren Rechenkapazität über mehrere Prozesse erreichen. Wenn eine Prozessausführung vor dem Ende ihrer Zeitscheibe terminiert, so verfällt der ungenutzte Zeitanteil, d.h. die nächste Ausführung des Prozessobjektes erhält eine neue Zeitscheibe, wenn sie beim nächsten Mal gestartet wird. Sowohl beim FIFO- als auch beim Zeitscheibenverfahren erfolgt die Planung ausschließlich aufgrund von Bereitzeiten. Fristen gehen nicht in die Betrachtung ein, und die Verfahren können keine Aussagen über die Einplanbarkeit von Prozessen machen, somit auch keine Garantie für eine fristgerechte Ausführung geben.

Sowohl das FIFO- als auch das Zeitscheiben-Verfahren sind –im Gegensatz zur vorgenannten zeitgesteuerten Zuteilung– sogenannte „arbeitserhaltende" Zuteilungsverfahren (engl.: *work-conserving*). Hiermit wird die Eigenschaft dieser Verfahren bezeichnet, die zu verfügbaren Betriebsmittel (hier: Rechenkapazitäten) vollständig an die Klienten (hier: Prozesse) zu verteilen, d.h. sofern mindestens ein rechenwilliger Prozess existiert ist der Prozessor unter einem arbeitserhaltenden Zuteilungsverfahren vollständig ausgelastet (vgl. [Chi07], S. 221).

Prioritätenbasierte Prozessorzuteilung

Die prioritätenbasierte Prozessorzuteilung ist ein in vielen Betriebssystemen anzutreffenes Verfahren zur impliziten Prozessplanung. Die Ergebnisse der Planung liegen dabei in Form von Prioritätswerten vor, die den Prozessen zugeordnet werden. Der Scheduler teilt den Prozessor dann jeweils demjenigen Prozess zu, der die höchste Priorität besitzt. Bei einigen Varianten wird vorausgesetzt, dass pro Prioritätswert nur ein Prozess existiert. Die meisten Implementierungen haben diese Einschränkung jedoch nicht, sodass ggf. mehrere rechenbereite Prozesse gleicher Priorität existieren können. Die Prozessorzuteilung zwischen solchen gleich priorisierten Prozessen geschieht nach dem FIFO- oder dem Zeitscheiben-Verfahren (siehe 2.2.8).

Wird ein Prozess rechenbereit, der eine höhere Priorität als der gerade laufende Prozess besitzt, so gibt es prinzipiell zwei mögliche Verfahrensweisen:

1. **Unterbrechende Prioritäten**: Ein arbeitender Prozess kann jederzeit von einem bereit werdenden Prozess höherer Priorität unterbrochen werden. Dies gilt auch, wenn Prozesse gleicher Priorität nach dem reinen FIFO-Verfahren, d.h. ohne begrenzte Zeitscheiben, arbeiten.

2. **Nicht unterbrechend**: Der arbeitende Prozess wird stets zu Ende geführt (engl.: *run to completion*), bevor der höher priorisierte Prozess den Prozessor erhält. Diese Vorgehensweise führt zu ähnlichen Problemen wie sie in 2.2.8 bereits für das FIFO-Verfahren genannt wurden: Lang arbeitende Prozesse können den Start anderer, höher priorisierter Prozesse sehr lange verzögern, d.h. das Zeitverhalten eines Prozesses wird stark vom Verhalten anderer Prozesse abhängig gemacht. Aus diesem Grund wird diese Zuteilungsmethode hier nicht weiter betrachtet.

Die Prioritäten können fest vergeben werden (engl: *fixed-priority scheduling*), oder sie können zur Laufzeit änderbar sein (engl: *dynamic priority scheduling*). Einige weit verbreitete Echzeit-Planungsverfahren (z.B. das Planen nach monotonen Raten [LL73]), liefern als Ergebnis eine feste Zuordnung von Prioritäten zu Prozessen. Diese Planungsverfahren können somit unmittelbar auf einem prioritätsbasierten Scheduler mit fester Prioritätsvergabe umgesetzt werden. Die meisten UNIX-Implementierungen arbeiten ebenfalls nach diesem Verfahren, wobei die Prioritäten der Prozesse durch das Betriebssystem dynamisch verändert werden: Durch das Warten eines Prozesses auf den Prozessor erhöht sich dessen Priorität allmählich; Prozesse, die den Prozessor intensiv nutzen, verlieren dabei allmählich an Priorität. Auf diese Weise soll eine „faire" Prozessorzuteilung erreicht werden, d.h. alle Prozesse kommen zum Zuge [Hen84, KL88, Ess90].

Der Laufzeitaufwand dieses Verfahrens lässt sich durch geschickte Implementierung ähnlich gering halten, wie bei den FIFO-basierten Verfahren. Insbesondere lässt sich das Verfahren so implementieren, dass seine Ausführungszeit unabhängig von der Anzahl zu betrachtender Prozesse ist, sodass für jede Ausführung des Schedulers immer von einer festen maximalen Dauer ausgegangen werden kann.

Auch die prioritätsgesteuerte Zuteilung ist arbeitserhaltend. Das Verfahren ist grundsätzlich auch auf Mehrprozessorsysteme anwendbar. Dazu wird, immer wenn ein Prozessor frei wird, dieser Prozessor dem jeweils höchstpriorisierten Prozess zugeteilt. Damit werden auf einem System mit N Prozessoren jeweils die N am höchsten priorisierten Prozesse parallel ausgeführt. Bei dieser Vorgehensweise besteht jedoch keine Möglichkeit, die Zugehörigkeit einzelner Teilprozesse zu „Task Forces" (vgl. 2.2.3) zu berücksichtigen: alle Prozesse werden als voneinander unabhängig betrachtet und gleich behandelt. Ein Coscheduling, wie in [Ous80] beschrieben, ist so nicht möglich.

Anteilige Prozessorzuteilung

Wie im vorigen Abschnitt erwähnt, versuchen UNIX-Systeme vielfach mit Hilfe prioritätsbasierter Prozessorzuteilung und dynamischer Prioritäten, eine „faire" Prozessorzuteilung zu erreichen, so, dass alle Prozesse an der zur Verfügung stehenden Prozessorleistung Anteil haben. Dabei ist es jedoch schwierig, den jeweiligen Anteil eines Prozesses genau vorzugeben (vgl. [NVZ01]). Die *anteilige Prozessorzuteilung* (engl.: *proportional share scheduling*) verfolgt ebenfalls das Ziel einer gleichmäßigen Verteilung von Rechenleistung, jedoch wird hier nicht der „Umweg" über Prioritäten verwendet, sondern den Prozessen werden unmittelbar Anteile an Rechenleistung zugeordnet und diese steuern direkt den Scheduler.

Erhalten alle Prozesse den gleichen Anteil an der Rechenleistung, so spricht man auch von *Processor Sharing*. Dieses stellt somit einen Spezialfall der anteiligen Prozessorzuteilung dar. Im allgemeinen Fall wird jedem Prozess i ein individuelles Gewicht w_i zugeordnet. Dieses Gewicht gibt den Anteil des betreffenden Prozesses an der gesamten Prozessorleistung an. Idealerweise wäre diese anteilige Zuteilung kontinuierlich, d.h. von jedem beliebigen Zeitintervall $[t_0, t_1]$ mit $t_0 < t_1$ würde der Prozess i einen Anteil von:

$$W_i(t_0, t_1) = (t_1 - t_0) \cdot \frac{w_i}{\sum_j w_j} \tag{2.9}$$

erhalten. Praktisch ist diese kontinuierliche Zuteilung nicht machbar, da ein Prozessor zu jedem Zeitpunkt immer nur vollständig einem einzigen Prozess zugeordnet sein kann. Bei der anteiligen Zuteilung wird versucht, die idealisierte Situation zu approximieren, indem den Prozessen entsprechend ihrer Gewichte mehr oder weniger häufig Zeitscheiben einer festen Länge zugeteilt werden. Diese Zeitscheibendauer wird im Zusammenhang mit der anteiligen Prozessorzuteilung auch als *Quantum*, q bezeichnet. Der Scheduler ermittelt nun für jeden Prozess einen *Rückstand* (engl: *lag*) als Differenz zwischen der Menge an Rechenzeit, die der Prozess in Form einzelner Quanten erhalten hat, und der Menge an Rechenzeit, die er nach Gleichung (2.9) idealerweise hätte erhalten sollen:

$$L_i(t_0, t_1) = n_i(t_0, t_1) \cdot q - (t_1 - t_0) \cdot \frac{w_i}{\sum_j w_j} \tag{2.10}$$

Darin ist $n_i(t_0, t_1)$ die Anzahl der Zeitscheiben, die dem Prozess i während des Zeitintervalls $[t_0, t_1]$ zugewiesen wurden. Ein positiver Rückstand zeigt an, dass der betreffende Prozess mehr als den ihm zustehenden Anteil an Rechenzeit erhalten hat, ein negativer Rückstand zeigt an, dass er weniger erhalten hat. Der Scheduler wird nach dem Ablauf eines jeden Quantums aufgerufen und wählt anhand

der aktuellen Rückstände einen Prozess aus der Prozessmenge aus, der für die Dauer des folgenden Quantums arbeiten kann. Beispielsweise kann stets derjenige Prozess gewählt werden, dessen Rückstand aktuell den niedrigsten (negativen) Wert hat. (Dabei kann es sich unter Umständen auch um den zuvor bereits aktiven Prozess handeln, d.h. nicht jeder Aufruf des Schedulers führt zwangsläufig zu einem Prozesswechsel.)

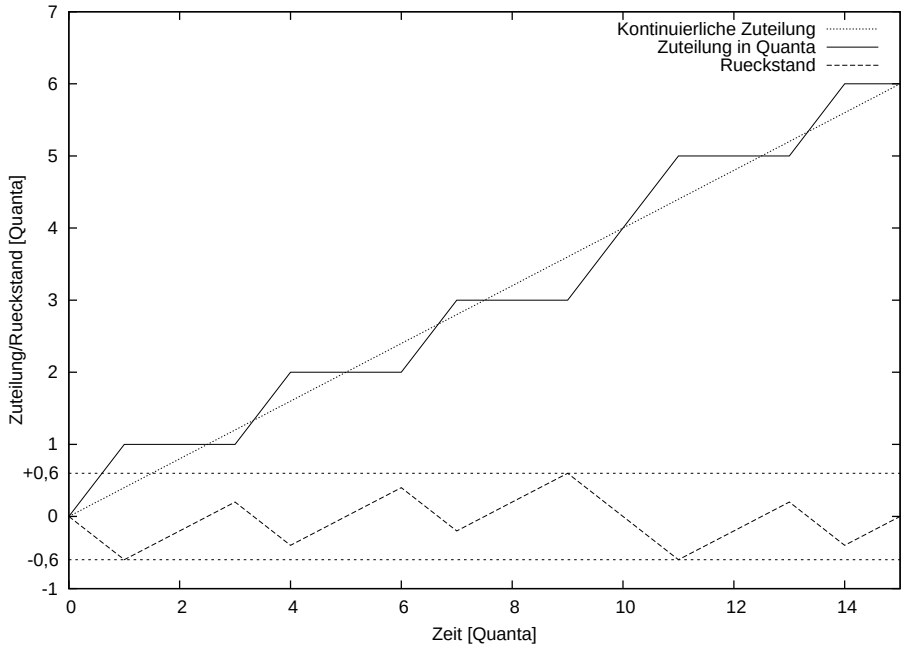

Abbildung 2.11: Kontinuierliche Zeitzuteilung und Zuteilung in Quanten.

Abbildung 2.11 zeigt als Beispiel den idealisierten (kontinuierlichen) Verlauf der Zeitzuteilung eines Prozesses mit einem Gewicht von 40% und die tatsächliche Zuteilung in Form einzelner Quanten nach diesem Verfahren. Ebenfalls gezeigt ist der sich ergebende Verlauf des Rückstandes, der bei diesem Beispiel auf ± 0.6 beschränkt ist.

Durch die anteilige Prozessorzuteilung wird alle verfügbare Rechenleistung auf die Prozesse verteilt, d.h. auch hier handelt es sich um ein arbeitserhaltendes Verfahren. Soll eine anteilige Zuteilung auf Basis eines prioritätsgesteuerten Schedulers durchgesetzt werden, so muss nach jedem abgelaufenen Quantum die Priorität eines Prozesses neu festgelegt werden. Wie bei EDF (vgl. 2.2.7) kann also auch hier vom „Planen mit dynamischen Prioritäten"gesprochen werden. Im Unterschied zu EDF kann hier aber auch eine Änderung der Priorität während der Dauer der Ausführung des Prozessobjektes stattfinden. Die Klasse von Planungsverfahren, auf die dies zutrifft, bezeichnen wir in dieser Arbeit auch als „uneinge-

schränkt dynamisch" (engl.: *unrestricted dynamic*, siehe [CFH+03]).

Als notwendige Bedingung muss der Scheduler sicherstellen, dass die Rückstände aller Prozesse zu jedem Zeitpunkt begrenzt sind. Der Absolutbetrag des größtmöglichen Rückstandes kann als Maß für die Qualität des Schedulers herangezogen werden. Ein weiteres wichtiges Kriterium zur Beurteilung eines Schedulers ist seine Laufzeitkomplexität: Da beispielsweise bei der beschriebenen Methode für jeden Prozess ein individueller Rückstandswert mitgeführt werden muss, ist ihr Rechenaufwand proportional zur Gesamtzahl der Prozesse. In der Literatur finden sich jedoch auch Verfahren mit konstantem Rechenaufwand (z.B. VTRR („virtual time round robin"), siehe [NVZ01]). Neben seiner Abhängigkeit von der Prozessanzahl ist natürlich auch der absolute Rechenaufwand relevant: Da in vielen Anwendungsfällen die Anzahl der Prozesse vorab festliegt, kann unter Umständen ein Verfahren, dessen Rechenaufwand mit steigender Prozessanzahl zunimmt, für eine gegebene Situation günstiger sein als eines, das von der Prozessanzahl zwar unabhängig ist, aber einen höheren Grundaufwand mit sich bringt. Ein drittes Kriterium ist der Aufwand bei Hinzukommen oder Wegfall eines Prozesses zur, bzw. aus der betrachteten Prozessmenge. In der Literatur werden etliche Verfahren beschrieben, die hinsichtlich dieser drei Kriterien unterschiedliche Präferenzen setzen ([Zha91, PG93, SV95, Wal95, SAWJ96, BZ96, GGV96, NVZ01]).

2.3 Virtualisierung

Unter *Virtualisierung* versteht man allgemein die Bereitstellung eines Modells eines Prozessors oder einer Systemarchitektur durch ein Programm. Ein solches Modell kann Programme ebenso ausführen, wie die Maschine, die es nachbildet. Es existieren verschiedene Formen der Virtualisierung, die sich hinsichtlich ihrer Zielsetzung, der Art der modellierten Maschine, sowie der zur Modellierung eingesetzten Techniken unterscheiden. Im Folgenden wird nach einer Abgrenzung zwischen nativer Virtualisierung und Virtualisierung durch Emulation ein an [Ros04] angelehnter, kurzer Überblick über verschiedene Arten der nativen Virtualisierung gegeben.

2.3.1 Emulation und native Virtualisierung

Eine weit verbreitete Form der Virtualisierung ist die Modellierung einer hypothetischen, also nicht physisch existierenden Architektur, wie sie sich beispielsweise bei den heutigen „Java Virtual Machines" findet ([LY99]) bzw. bereits bei den „p-"

und „m-Maschinen" Anfang der 80er Jahre fand ([Bow78]). Diese Systeme führen Code für eine gedachte Rechnerarchitektur aus, indem sie ihn interpretieren. Der Code[1] ist dadurch plattformunabhängig, d.h. er kann auf jeder Maschine ausgeführt werden, für die es einen entsprechenden Interpreter gibt. Hierin liegt auch die Zielsetzung dieses Ansatzes: Anstatt beim Wechsel zu einer anderen Rechnerplattform sämtliche Programme neu zu übersetzen, wird nur ein einziges Programm, der Interpreter, neu übersetzt. Alle weiteren Programme können dann auf der neuen Plattform unverändert laufen, ohne dass ihr Quellcode dazu verfügbar sein müsste.

In gleicher Weise wie eine hypothetische Maschine kann auch eine real existierende Maschine emuliert werden. Beispielsweise ist „Bochs" ([Law96]) ein Programm, das das Verhalten eines typischen PC so detailgetreu nachbildet, dass darauf beliebige Betriebssysteme und Anwenderprogramme wie auf einem realen PC arbeiten können. Die Programmausführung durch Interpretation einzelner Maschinenbefehle mit Hilfe eines solchen Emulators ist, gemessen an der Programmausführung durch eine reale Maschine, zwangsläufig um Größenordnungen langsamer. Allerdings ist es damit möglich, Programme, die für eine bestimmte Rechnerarchitektur übersetzt wurden, ohne jegliche Änderung auf einer anderen Rechnerarchitektur zu verwenden. Außerdem ist die emulierte Maschine bis ins Detail kontrollierbar: sie kann an beliebiger Stelle angehalten werden, ihre Registerinhalte können jederzeit inspiziert und verändert werden usw., sodass sie eine sehr nützliche Umgebung für das Entwickeln und Testen hardwarenaher Programme darstellt. Das Entwickeln und Testen von Betriebssystemsoftware ist dementsprechend eines der Haupt-Einsatzgebiete für Programme wie „Bochs".

Aufgrund der erheblichen Leistungseinbußen durch die Interpretation des Codes ist diese Form der Virtualisierung durch Emulation im Kontext dieser Arbeit nicht weiter von Interesse. Falls aber der Rechner, der das Modell ausführt, den Befehlssatz des modellierten Rechners direkt unterstützt, und falls er zudem einige weitere Kriterien erfüllt [PG74], so kann der Wirtsprozessor die meisten Befehle des Zielprozessors direkt abarbeiten. In diesem Fall können Programme ohne große Leistungseinbußen in dem Modell ausgeführt werden. Diese Form der „nativen" Virtualisierung soll im Folgenden eingehender betrachtet werden. Wenn im Rest der Arbeit von „Virtualisierung" gesprochen wird, so ist damit stets die native Virtualisierung gemeint.

[1] Bei Java Virtual Machines wird er als *Bytecode* bezeichnet.

2.3.2 Virtuelle Maschinen

Die Idee der Virtualisierung entstand bereits Ende der 60er Jahre bei IBM [Cre81]. Damals war das Ziel, teure Mainframe-Rechner durch Zeitmultiplex-Betrieb so zu vervielfachen, dass sie gleichzeitig für unterschiedliche Aufgaben genutzt werden konnten. IBM definierte dazu die *virtuelle Maschine* als eine vollständig isolierte Kopie der unterlagerten physischen Maschine. Anwendungen und sogar Betriebssysteme sollten sich bei der Ausführung durch eine virtuelle Maschine exakt so verhalten, als würden sie von einer realen Maschine ausgeführt.

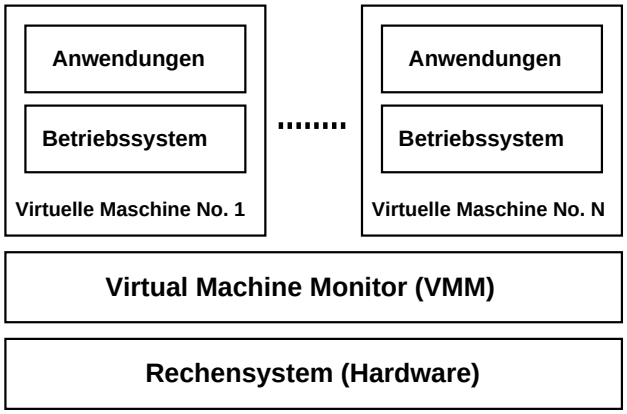

Abbildung 2.12: Virtual Machine Monitor (VMM) und Gastsysteme.

Die Softwarekomponente, die eine virtuelle Maschine implementiert, wird als *Virtual Machine Monitor* (VMM), mitunter auch als *Hypervisor* bezeichnet (siehe Abbildung 2.12). Dieser Monitor bietet eine Abstraktion der Maschine, die als solche mehrfach instanziierbar ist. Auf diese Weise lassen sich auf einer einzigen physischen Maschine mehrere virtuelle Maschinen implementieren, in denen gleichzeitig mehrere, potenziell unterschiedliche Programme und Betriebssysteme auf einer gemeinsamen Hardware ausgeführt werden können. Im Kontext der Virtualisierung werden diese Programme und Betriebssysteme oft auch als *Gäste* (engl.:*guest*) bezeichnet. Die Verteilung der physischen Betriebsmittel der unterlagerten Rechnerarchitektur auf die verschiedenen virtuellen Maschinen obliegt dem Virtual Machine Monitor. Diese Betriebsmittel umfassen neben Hauptspeicher und Ein-/Ausgabegeräten auch Rechenzeit, d.h. Häufigkeit und Dauer von Zeitintervallen, während derer eine virtuelle Maschine den physischen Prozessor (bzw. die physischen Prozessoren) des Rechensystems nutzen kann. Werden diese Betriebsmittel so verteilt, dass jeder Gast über eine eigene Untermenge davon exklusiv verfügt, so lässt sich sicherstellen, dass kein Gast Betriebsmittel eines anderen Gastes (mit)benutzen kann. Auf diese Weise lassen sich die Gäste ebenso

vollständig voneinander isolieren, als würden sie auf jeweils eigenen physischen Rechnern arbeiten.

In den folgenden Abschnitten werden für die genannten Arten von Betriebsmitteln jeweils Techniken vorgestellt, die ihre Aufteilung auf verschiedene virtuelle Maschinen eines gemeinsamen Rechners ermöglichen.

2.3.3 Prozessorvirtualisierung

Wie bereits angedeutet, befasst sich diese Arbeit ausschließlich mit Techniken zur nativen Virtualisierung. Dabei geht es an erster Stelle darum, ein effizientes Modell der Programmausführung zu schaffen. Dieses soll möglichst große Teile der auszuführenden Programme direkt „in Hardware" auf dem Prozessor des Wirtsrechners abarbeiten.

In [PG74] definieren Popek und Goldberg formale Anforderungen, die eine Rechnerarchitektur erfüllen muss, damit sie in diesem Sinne „virtualisierbar" ist. Die wichtigste Anforderung ist dabei, dass:

- *die Menge der sensitiven Maschinenbefehle eine Untermenge der privilegierten Maschinenbefehle darstellt.*

Dabei sind „sensitive" Maschinenbefehle solche Instruktionen, mit denen die Betriebsmittelkonfiguration der Maschine verändert werden kann, oder deren Ergebnis von dieser Konfiguration abhängt. So zählt beispielsweise das Manipulieren, aber auch bereits das reine Lesen einer Interruptmaske zu den sensitiven Operationen. Mit „privilegierten" Instruktionen sind solche gemeint, deren Ausführung im nicht-privilegierten Modus zu einer Ausnahmebedingung („Trap") führt. Einfacher ausgedrückt besagt das Kriterium also, dass sämtliche sensitiven Maschinenbefehle eine Ausnahmebedingung zur Folge haben müssen, wenn sie im nicht-privilegierten Modus ausgeführt werden. Eine solche Ausnahmebedingung kann vom Virtual Machine Monitor abgefangen und behandelt werden, sodass er den privilegierten Befehl emulieren kann, wobei aber die Betriebsmittelkonfiguration der virtuellen Maschine, nicht die der realen Maschine, verändert bzw. ermittelt wird. Dieser Vorgang bleibt für das unterbrochene Programm unsichtbar.

Popek und Goldberg geben drei wesentliche Charakteristika eines Virtuellen Machine Monitors an:

1. *Ein Programm, das als Gast unter einem Virtual Machine Monitor arbeitet, muss gegenüber dem gleichen Programm, das auf der entsprechenden*

physischen Maschine ausgeführt wird, identisches Verhalten zeigen. Hierbei wird jedoch eine Ausnahme zugelassen: Das Zeitverhalten des Gastes kann sich von dem seines auf der physischen Maschine arbeitenden Äquivalentes unterscheiden. Damit deutet sich bereits an, dass zur Anwendung der Virtualisierung auf Aufgabenstellungen aus dem Bereich der Echtzeitverarbeitung weitere Betrachtungen angestellt werden müssen.

2. *Ein Programm, das als Gast unter einem Virtual Machine Monitor arbeitet, zeigt gegenüber dem gleichen Programm, das auf der entsprechenden physischen Maschine ausgeführt wird, im ungünstigsten Fall eine nur geringfügig schlechtere Leistung.* Dazu ist es nötig, dass der im statistischen Mittel überwiegende Anteil der zur Programmierung verwendeten Instruktionen direkt auf dem realen Prozessor ausgeführt werden kann. Virtualisierung kann nur dann effizient sein, wenn die Behandlung von „Traps" zur Emulation privilegierter Befehle eine seltene Ausnahme darstellt.

3. *Der Virtual Machine Monitor muss die uneingeschränkte Kontrolle über alle Betriebsmittel ausüben.* Da dem Virtual Machine Monitor die Aufteilung der Betriebsmittel obliegt, müssen sämtliche Zugriffe auf physische Betriebsmittel von ihm kontrolliert werden.

Die erste Implementierung einer Virtualisierung arbeitete auf einer IBM 360/67 Maschine. Da ihre Leistung eher enttäuschend ausfiel, konstruierte man bei IBM als Nachfolger die „Virtual Machine Facility/370" (VM/370) ([Gum83]), deren Befehlssatz speziell auf die Erfordernisse einer effizienten Virtualisierbarkeit zugeschnitten wurde.

Vor allem im Desktop-Bereich, mittlerweile aber auch im Server-Bereich, ist heute die IA-32-Architektur von Intel die wohl am weitesten verbreitete Rechnerarchitektur. Sie wurde ursprünglich nicht für die Virtualisierung konzipiert. Insbesondere enthält ihr Befehlssatz insgesamt 17 sensitive Instruktionen, die im nichtprivilegierten Modus ausgeführt werden können, ohne dass dabei eine Ausnahmebehandlung ausgelöst wird ([RI00]). Dies verletzt die oben zitierte Hauptanforderung von Popek und Goldberg, weshalb die IA-32-Architektur lange Zeit als nicht virtualisierbar galt. Durch „kreatives Programmieren" gelang es aber schließlich doch, auch auf dieser Architektur virtuelle Maschinen zu implementieren. In [RI00] geben Robin und Irvine die üblicherweise verwendete Methode an. Dabei werden die sensitiven, nicht privilegierten Instruktionen im Code eines neu geladenen Programms vor dessen Ausführung lokalisiert (dies wird als *Scan before execution* oder *Prescan* bezeichnet) und durch entsprechende andere Befehle substituiert, die eine Ausnahmebedingung und damit einen Aufruf des Virtual Machine Monitors bewirken.

Durch die neue Anwendbarkeit auf die IA-32-Architektur ist die Virtualisierung nun auch im Desktop-Bereich einsetzbar. Neben der bereits genannten Motivation, einen Rechner mehrfach nutzen zu können, wird die Virtualisierung hier vor allem auch dazu genutzt, mehrere Betriebssysteme auf einem Rechner zu betreiben und so Anwendungen, die für unterschiedliche Betriebssystemschnittstellen entwickelt wurden, gemeinsam auf einem Rechner nutzen zu können. Die Technik der Virtualisierung hat dadurch in letzter Zeit viel neue Aufmerksamkeit erhalten. Diese neue Aufmerksamkeit hat wiederum dazu geführt, dass die beiden großen Hersteller von IA-32-Prozessoren, Intel und AMD, mit ihren „Vanderpool-" bzw. „Pacifica-" Architekturen ([MUKX06]) mittlerweile neue, „virtualisierungsfreundliche" Varianten ihrer Prozessoren anbieten, mit denen nun auch eine vollständige Virtualisierung ohne Kunstgriffe wie den erwähnten Prescan möglich ist.

Voll- und Paravirtualisierung

Die bisher beschriebene Form der Virtualisierung wird auch als *Vollvirtualisierung* bezeichnet. Sie ist –insbesondere auf Architekturen wie IA-32, die den Anforderungen von Popek und Goldberg nicht genügen– schwierig zu implementieren. Die Notwendigkeit, jedes Programm vor seiner Ausführung erst nach sensitiven, nicht-privilegierten Instruktionen durchsuchen zu müssen, führt bei dieser Technik zu deutlichen Leistungseinbußen. Allerdings haben diese Lösungen den großen Vorteil, dass sie Programme und Betriebssysteme, die für die Wirtsarchitektur entwickelt wurden, ohne jegliche Veränderung ausführen können, d.h. sie sind vollständig „binärkompatibel". Ein anderer Ansatz ist hier die *Paravirtualisierung*. Dieser Begriff wurde von Gribble et al. mit dem Virtual Machine Monitor „Denali" eingeführt [WSG02]. Bei der Paravirtualisierung wird der Anspruch der Binärkompatibilität –zumindest für den Betriebssystemcode– aufgegeben: Der Code des Gast-Betriebssystems, also der Teil des Gastes, der auf einem physischen System im privilegierten Modus ablaufen würde, wird so verändert, dass er keine sensitiven Instruktionen mehr verwendet. Stattdessen werden entsprechende Aufrufe in den Virtual Machine Monitor (sogenannte *Hypercalls*) ausgeführt. Diese Aufrufe treten logisch an die Stelle der Ausnahmebehandlung bei der Vollvirtualisierung. Eine solche Veränderung des Gastsystems erfolgte bisher üblicherweise auf Quellcode-Ebene. Neuere Ansätze ([LUC+05]) arbeiten hingegen mit speziellen Compilern, bzw. Assemblern, die bereits während der Übersetzung die sensitiven Instruktionen durch geeignete Hypercalls substituieren. Auf diese Weise ist prinzipiell kein Eingriff in den Quellcode des Gastsystems mehr erforderlich, allerdings muss er für die Übersetzung zur Verfügung stehen, sodass auch mit dieser Methode keine Betriebssysteme virtualisiert werden können, de-

ren Quellcode nicht vorliegt.

Die Änderungen am Gastsystem betreffen ausschließlich den privilegierten Teil, d.h. in der Regel den Betriebssystemkern. Anwenderprogramme, die auf der physischen Maschine nicht im privilegierten Modus arbeiten, können sowohl unter einer Paravirtualisierung als auch unter einer Vollvirtualisierung unverändert weiter verwendet werden.

2.3.4 Arbeitsspeichervirtualisierung

Neben einem virtuellen Prozessor muss eine virtuelle Maschine auch alle weiteren Bestandteile eines physischen Rechners nachbilden. Um den virtuellen Maschinen Arbeitsspeicher zur Verfügung stellen zu können, nutzt ein Virtual Machine Monitor die gleiche Technik des virtuellen Speichers, die auch von klassischen Mehrbenutzerbetriebssystemen bekannt ist. Das in [PG74] verwendete Modell eines Prozessors enthält dazu ein *Relocation Bound Register*. Dieses Register ist Bestandteil der virtuellen Maschine und es beschreibt einen Speicherbereich durch Basisadresse und Länge. Versucht eine virtuelle Maschine, eine Adresse außerhalb dieses Bereiches zu referenzieren, so wird eine Ausnahmebedingung erzeugt. Bei Adressen innerhalb des Bereiches wird die physische Adresse durch Addition des Basisadresse gebildet. Auf diese Weise kann der Virtual Machine Monitor den physischen Speicher in Segmente unterteilen, die er jeweils verschiedenen virtuellen Maschinen zuordnet, und er kann verschiedene physische Segmente für verschiedene virtuelle Maschinen an gleichen Adressen erscheinen lassen.

Das Relocation Bound Register stellt eine vereinfachte Abstraktion der Mechanismen dar, die heutige Rechnerarchitekturen in Form einer Speicherverwaltungseinheit (MMU) bieten. Die Konfiguration der Adressabbildung erfolgt dabei in der Regel durch eine Seitentabelle (eine im Hauptspeicher abgelegte Datenstruktur), die somit auch zur Betriebsmittelkonfiguration der virtuellen Maschine zählt. Der Zugriff auf Seitentabellen gehört damit zu den sensitiven Operationen.

Die mit Hilfe der Speicherverwaltungseinheit implementierten virtuellen Adressräume können aus Sicht des Gastsystems wiederum als physische Adressräume aufgefasst werden. Darauf basierend können nach dem Prinzip der *rekursiven Adressräume* ([LRD95, Lie95]) auf Gastsystem-Ebene neue virtuelle Adressräume als Laufzeitumgebung für Anwenderprogramme geschaffen werden. Prinzipiell kann diese Rekursion beliebig tief verschachtelt werden. In den meisten praktischen Fällen wird jedoch nur eine zusätzliche Ebene verwendet: Der Virtual Machine Monitor bietet seinen Gastsystemen eine kontrollierte Zugriffsmöglichkeit auf die Teile der Seitentabelle, die jeweils deren eigene Adressräume betreffen,

die Gastsysteme exportieren jedoch für gewöhnlich keine derartige Schnittstelle an ihre Anwendungen.

2.3.5 Ein-/Ausgabe-Virtualisierung

Ein virtuelle Maschine, die das Verhalten eines vollständigen Rechensystems modellieren soll, muss neben Prozessor und Hauptspeicher auch die Peripheriekomponenten dieses Rechensystems abbilden. Gerade Betriebssysteme enthalten in der Regel auch Gerätetreiber, die mit den Ein-/Ausgabebausteinen des Rechners interagieren müssen. Sollen solche Betriebssysteme als Gäste in einer virtuellen Maschine arbeiten, so muss diese einerseits die vom Betriebssystem bzw. den Gerätetreibern erwarteten Schnittstellen bereitstellen, und andererseits muss sie die Funktionalität der Periperhiekomponenten nachbilden.

Die Programmierung von Peripheriegeräten durch Gerätetreiber geschieht über entsprechende Lese- und Schreibzugriffe auf die Register einer solchen Hardware. Diese Register liegen als spezielle Speicherstellen im Speicheradressraum des Rechners (*memory-mapped I/O*), oder sie liegen im E/A-Adressraum, wo sie mit speziellen Maschinenbefehlen gelesen oder geschrieben werden können (*port-mapped I/O*). In beiden Fällen kann durch entsprechende Programmierung der Speicherverwaltungseinheit (MMU) des Prozessors für jedes Peripheriegerät individuell festgelegt werden, ob Zugriffe auf seine Register privilegierte Operationen darstellen oder nicht. Hierbei kann es jedoch hardware-bedingte Einschränkungen geben: Zugriffsrechte auf memory-mapped I/O-Bereiche können maximal mit der Granularität einer Speicherseite (typischerweise 4KB) vergeben werden.

- Sind E/A-Registerzugriffe nicht privilegiert, so können Programme, die in einer virtuellen Maschine arbeiten, direkt auf die Hardware der Peripheriebausteine zugreifen. Der Virtual Machine Monitor muss dann sicherstellen, dass jedes Perpheriegerät genau einer virtuellen Maschine zugeordnet wird, d.h. dass seine Register nur im Adressraum *einer* virtuellen Maschine zugreifbar sind. In diesem Fall „gehört" das jeweilige Peripheriegerät exklusiv dieser virtuellen Maschine. Es ist damit zwar nicht von mehreren virtuellen Maschinen nutzbar, aber aufgrund seiner exklusiven Zuordnung zu genau einer virtuellen Maschine bleibt die Isolation der virtuellen Maschinen erhalten. Wir bezeichnen diese Vorgehensweise als *E/A-Partitionierung*.
- Sind E/A-Registerzugriffe privilegiert, so löst jeder Versuch eines Zugriffs eine Ausnahmebedingung aus, die einem Virtual Machine Monitor Gelegenheit gibt, eine Emulation des (möglicherweise physisch nicht vorhandenen) Peripheriebausteins einzublenden. Aus der Sicht der virtuellen Maschine ist diese Emulation nicht von dem entsprechenden physischen Baustein

zu unterscheiden. Je nach den Eigenheiten des nachgebildeten Bausteins kann eine solche Emulation eine komplexe Aufgabe darstellen. Zudem ist die Behandlung der häufig auftretenden Ausnahmebedingungen deutlich teurer als es die damit emulierten Ein-/Ausgabeoperationen auf der physischen Maschine wären. Die Emulation ist also eine eher schwerfällige Schnittstelle. Ihr unbestrittener Vorteil liegt aber, wie bereits bei der Vollvirtualisierung, in der vollständigen Transparenz der Virtualisierung aus der Sicht des Gastsystems, d.h. auch die Gerätetreiber eines Gastsystems können damit unverändert in der virtuellen Maschine arbeiten. In Analogie zur bereits eingeführten (Prozessor-)Vollvirtualisierung bezeichnen wir diese Methode deshalb als *E/A-Vollvirtualisierung*. Systeme, die eine Vollvirtualisierung anstreben, verwenden in der Regel diese Technik zur Abbildung von Ein-/Ausgabebausteinen.

- Anstatt Peripheriebausteine aufwändig zu emulieren, kann ein Virtual Machine Monitor für seine virtuellen Maschinen auch eine spezielle Schnittstelle zur Peripherie bereitstellen. Dabei werden die Funktionen der Peripheriebausteine direkt durch Hypercalls aufgerufen. Die Funktionalität dieser Hypercalls kann sich an der bereitzustellenden Funktionalität des Peripheriegerätes orientieren, anstatt sich mit den Eigenheiten bestimmter Ein-/Ausgabebausteine auseinanderzusetzen. Beispielsweise kann mit einem einzigen Hypercall ein ganzer Block Daten auf eine Festplatte geschrieben werden. Um den gleichen Effekt mit Hilfe einer Emulationsschnittstelle zu erzielen, müssten etliche Ausnahmebedingungen abgefangen und behandelt werden. Analog zur (Prozessor-)Paravirtualisierung ist diese Art der Virtualisierung von Ein-/Ausgabegeräten also effizienter als die E/A-Vollvirtualisierung, aber gleichzeitig macht sie eine Anpassung der Gerätetreiber des Gastsystems erforderlich. Wir bezeichnen sie deshalb als *E/A-Paravirtualisierung*.

Die genannten Techniken sind auch gemischt nutzbar, d.h. während einzelne Peripheriegeräte exklusiv bestimmten virtuellen Maschinen zugewiesen werden können, ist es gleichzeitig möglich, andere Peripheriebausteine zu emulieren und zudem Hypercalls zur Nutzung durch spezielle Gerätetreiber bereitzustellen.

Unabhängig davon, ob die Schnittstelle durch Emulation oder über Hypercalls angesprochen wird, benötigt der Virtual Machine Monitor Zugriff auf reale Peripherie, um die Funktionalität der virtuellen Peripherie realisieren zu können. Soll beispielsweise mehreren virtuellen Maschinen jeweils eine eigene Festplatte angeboten werden, so muss der Virtual Maschine Monitor auf einen realen Massenspeicher zugreifen können, um die von den virtuellen Maschinen geschriebenen

oder gelesenen Daten speichern zu können. Er kann dazu den realen Massenspeicher in verschiedene Bereiche („Partitionen") aufteilen, auf die er jeweils eine der virtuellen Festplatten abbildet. Jede virtuelle Maschine erhält so exklusiven Zugriff auf einen Teil der physischen Festplatte. Auf diese Weise kann ein nur einfach vorhandenes physisches Peripheriegerät von mehreren virtuellen Maschinen gemeinsam genutzt werden.

Analog dazu kann auch eine nur einfach vorhandene Netzwerkschnittstelle für mehrere virtuelle Maschinen derart nutzbar gemacht werden, dass jede von ihnen scheinbar über eine eigene Schnittstelle verfügt. Dazu werden vom Virtual Machine Monitor beim Versenden von Paketen abhängig davon, welche der virtuellen Maschinen ein Paket verschickt, jeweils verschiedene Hardware- („MAC-") Adressen verwendet. Für andere Teilnehmer des Netzwerkes entsteht dadurch der Eindruck, als stammten die Pakete von verschiedenen physischen Netzwerkschnittstellen. Empfangsseitig besteht z.B. die Möglichkeit, die Schnittstelle in den eigentlich zur Netzwerkanalyse gedachten, sogenannten „promiscuous mode" zu bringen. Dabei werden sämtliche Pakete des Netzwerks von der Schnittstelle empfangen, unabhängig davon, an welchen Teilnehmer sie adressiert sind. Aufgrund der in den Paketen eingetragenen Zieladressen kann der Virtual Machine Monitor dann entscheiden, ob ein Paket einer seiner virtuellen Maschinen zugestellt werden muss.

2.3.6 Zeit-Virtualisierung

Wie in Abschnitt 2.3.3 bereits kurz erwähnt, haben Popek und Goldberg in [PG74] das Zeitverhalten virtueller Maschinen explizit von ihren Überlegungen ausgeschlossen. Es findet sich lediglich der pauschale Hinweis, dass sich das Zeitverhalten einer virtuellen Maschine von dem einer physischen Maschine unterscheidet. Dafür gibt es mehrere Ursachen:

1. Die Behandlung von Ausnahmebedingungen und die Emulation privilegierter Instruktionen durch den Virtual Machine Monitor ist in der Regel teurer als das direkte Ausführen der Instruktionen im privilegierten Modus.

2. Der erwähnte „Prescan", der bei Architekturen wie IA-32 vor der Ausführung von Programmen vorgenommen werden muss, ist mit zusätzlichen Kosten verbunden.

3. Wenn mehrere virtuelle Maschinen auf einer physischen Maschine arbeiten, so kann ein physischer Prozessor zu jedem Zeitpunkt immer nur von

einer virtuellen Maschine genutzt werden. Die virtuellen Maschinen konkurrieren also um die physischen Prozessoren, und die Prozessoren stehen den virtuellen Maschinen nur für einen (vom Virtual Machine Monitor zu kontrollierenden) Teil der real verstreichenden Zeit zur Verfügung.

4. Die Kontextwechsel der Prozessoren bei der Umschaltung zwischen den virtuellen Maschinen sind mit zusätzlichen Kosten verbunden.

Die beiden erstgenannten Ursachen sind unvermeidliche Eigenschaften der Virtualisierung. Der daraus resultierende Einfluss auf das Zeitverhalten besteht in einer generellen Leistungseinbuße gegenüber der physischen Maschine, deren Umfang allein durch die relative Häufigkeit privilegierter Befehle im Code des Gastprogramms bestimmt ist. Diese Leistungseinbuße hängt vom Code des Gastes und von der angewendeten Virtualisierungstechnik (D.h.: mit oder ohne Prescan) ab, und sie ist insbesondere unabhängig vom Verhalten anderer virtueller Maschinen, die auf derselben physischen Maschine arbeiten. Diese Verluste fallen kontinuierlich während der Ausführung des Gastprogramms bzw. einmalig vor dem Beginn der Ausführung an. Sie ändern sich zur Laufzeit nicht und können damit relativ leicht bei einer Echtzeitplanung berücksichtigt werden.

Anders verhält es sich mit den beiden letztgenannten Ursachen: Üblicherweise schaltet ein Virtual Machine Monitor zyklisch zwischen den unter ihm arbeitenden virtuellen Maschinen um. Jede von ihnen ist demnach nur während eines zyklisch wiederholten Zeitintervalls aktiv, d.h. jede virtuelle Maschine „sieht" nur einen Ausschnitt aus der real verstreichenden Zeit.

Die Häufigkeit der Umschaltungen und die Länge der Zeitintervalle hängen vom Zuteilungsverfahren des Virtual Machine Monitors sowie –bei vielen existierenden Implementierungen– auch vom Verhalten der Gäste anderer virtueller Maschinen desselben Virtual Machine Monitors ab. Für die bisherigen Anwendungsfälle im Server- und Desktop-Bereich ist in der Regel nur der mittlere Systemdurchsatz einer virtuellen Maschine relevant, weshalb hier bereits einfache Angaben über die durchschnittlich zugeteilte Rechenzeit ausreichen. Auch gelegentliche Abweichungen von diesen Mittelwerten sind tolerierbar, solange sie nicht zu allzu häufig und nicht über ausgedehnte Zeiträume hinweg auftreten.

In dieser Arbeit sollen nun auch Gäste mit harten Zeitanforderungen betrachtet werden. Hier sind Durchschnittsangaben über die zugeteilte Rechenzeit nicht mehr ausreichend. In der Literatur finden sich keine Betrachtungen solcher Anwendungsszenarien virtueller Maschinen. Diese Thematik soll in Kapitel 4 eingehend untersucht werden.

2.3.7 Beispiele: VMWare und Xen

Die beiden Virtualisierungslösungen VMWare ([SVL01]) und Xen ([BDF+03]) haben in den letzten Jahren neue Aufmerksamkeit auf das Konzept der Virtualisierung gelenkt. Diese beiden Systeme werden hier zunächst vorgestellt und anhand der bisher vorgestellten Kriterien eingeordnet. In Kapitel 4 wird, nachdem die nötigen Modellerweiterungen in Kapitel 3 eingeführt sind, näher auf ihr Zeitverhalten eingegangen.

VMWare

VMWare ist eine populäre kommerzielle Virtualisierungslösung für die IA-32-Architektur, die einen Standard PC einschließlich einer Auswahl typischer Ein/-Ausgabegeräte vollständig virtualisiert. In ihren virtuellen Maschinen können beliebige Betriebssysteme mit ihren Anwenderprogrammen unmodifiziert ablaufen. Es existieren mehrere Varianten:

- Die Version „VMWare Workstation" arbeitet als Anwenderprozess unter einem vollen „Wirts-" Betriebssystem wie Linux oder Windows und nutzt dessen Gerätetreiber- Infrastruktur, um auf Netzwerk, Massenspeicher etc. zuzugreifen. Neben dem IA-32 Prozessor bildet VMWare die vollständige Ausstattung an Ein-/Ausgabegeräten eines typischen PC nach: Jede virtuelle Maschine hat Zugriff auf Tastatur, Maus, Disketten-, Festplatten- und CD-ROM-Laufwerke, etc., sowie auf eine beliebige Anzahl von Netzwerk-Schnittstellen, die den weitverbreiteten AMD PCNet Chipsatz detailgetreu nachbilden.

  Zusätzlich zu dem Anwenderprozess verwendet VMWare Workstation auch optional einen speziellen Gerätetreiber, den *VMDriver*, der den virtuellen Maschinen einen effizienteren Zugriff auf die Geräte des Wirtssystems ermöglicht.

  Die Netzwerkschnittstelle von VMWare versetzt mit Hilfe des VMDriver die physische Schnittstelle, wie in 2.3.5 beschrieben, in den „promiscuous mode", wodurch VMWare mit nur einer einzigen physischen Netzwerkschnittstelle jeder virtuellen Maschine eine eigene virtuelle Netzwerkschnittstelle zur Verfügung stellen kann.

- Die Version „VMWare ESX Server" arbeitet ohne ein unterlagertes Betriebssystem direkt auf der Hardware des Wirtsrechners und muss daher seine eigene Gerätetreiber-Infrastruktur zum Zugriff auf Netzwerk, Massenspeicher, etc. mitbringen. Die Funktionalität der virtuellen Maschine von

VMWare ESX Server ist vergleichbar mit der von VMWare Workstation. Dieses System entspricht dem klassischen Bild der Virtualisierung: ein Virtual Machine Monitor, der als einziges Element im privilegierten Modus arbeitet.

Beide Varianten von VMWare implementieren sowohl Prozessor- als auch E/A-Vollvirtualisierung für IA-32 Rechner, wobei sensitive Instruktionen nach der in 2.3.3 erwähnten „Prescan"-Methode dynamisch substituiert werden. Damit kann VMWare beliebige Betriebssysteme für IA-32 PCs unmodifiziert in seinen virtuellen Maschinen ausführen. Ein Gastsystem kann, sofern es in dieser Hinsicht von VMWare unterstützt wird, spezielle Gerätetreiber einbinden, die den virtuellen Maschinen durch E/A-Paravirtualisierung noch effizienteren Zugang zu den physischen Geräten des Wirtssystems erlauben.

Xen

„Xen" ist ein Virtual Machine Monitor für Intel IA-32 Architekturen. Im Unterschied zu VMware ESX Server realisiert Xen eine *Para*virtualisierung. Das System wurde an der Universität von Cambridge (Großbritannien) entwickelt und verfolgt das Ziel, gängige Betriebssysteme wie Windows, Linux oder NetBSD als mögliche Gastsysteme zu unterstützen, wobei die Effizienz des Systems nach [BDF+03] über der von VMWare Workstation liegen soll[1]. Im Gegensatz zu VMWare ist bei Xen der gesamte Quellcode des Systems frei verfügbar. Da es sich um eine Paravirtualisierungsumgebung handelt, müssen Gastbetriebssysteme an Xen angepasst werden, damit sie in seinen virtuellen Maschinen arbeiten können. Anwenderprogramme können jedoch unverändert weiterverwendet werden. Zur Virtualisierung der Ein-/Ausgabe arbeitet Xen sowohl mit E/A-Partitionierung als auch mit E/A-Paravirtualisierung: Die „Domain0", eine der unter Xen arbeitenden virtuellen Maschinen, ist dadurch ausgezeichnet, dass der in ihr arbeitende Gast uneingeschränkten Zugriff auf die Register der Ein-/Ausgabebausteine des Wirtsrechners hat. Für alle weiteren virtuellen Maschinen bietet Xen virtuelle Ein-/Ausgabegeräte an, die über Hypercalls angesprochen werden können. Die Dienstanforderungen, die die virtuellen Maschinen an diese virtuellen Geräte richten, werden von Xen an die Domain0 weitergeleitet, wo sie auf die dieser Domain zugänglichen physischen Geräte abgebildet werden. Auf diese Weise benötigt auch Xen keine eigenen Gerätetreiber, sondern es kann stattdessen die Treiber des Gastsystems in Domain0 nutzen. Nachteilig ist dabei allerdings, dass alle übrigen virtuellen Maschinen, wie auch Xen selbst, vom korrekten Funktionieren des Gastbetriebssystems in Domain0 abhängen.

[1] Nach [VMw07] ist wiederum VMWare ESX Server effizienter als Xen.

Um den Aufwand bei der Anpassung bestehender Betriebssysteme an Xen möglichst gering zu halten, verwalten Gastbetriebssysteme ihre Seitentabellen zur Programmierung der MMU selbst. Deren Code zur Speicherverwaltung kann deshalb nahezu unverändert bestehen bleiben. Um dennoch eine sichere Isolation mehrerer koexistierender virtueller Maschinen gewährleisten zu können, sind die Seitentabellen für die Gastbetriebssysteme nur lesbar. Ein Versuch, sie zu schreiben, führt –für das Gastsystem unsichtbar– zu einer Ausnahmebedingung, die dem Virtual Machine Monitor Gelegenheit gibt, die Korrektheit des zu schreibenden Datums zu validieren, bevor er es tatsächlich in die Seitentabelle einträgt.

Bemerkung: Aufgrund der freien Verfügbarkeit des Quellcodes von Xen, und da dieses System den Austausch des standardmäßig eingesetzten Schedulers unterstützt, wird es auch als mögliche Experimentalplattform zur praktischen Erprobung der zu entwickelnden Techniken zur Echtzeitverarbeitung ins Auge gefasst.

Kapitel 3

Modellbildung

Nachdem im vorangegangen Kapitel 2 die Grundlagen zu den drei Gebieten dieser Arbeit zusammengestellt wurden, werden in diesem Kapitel eigene Modelle zur Erweiterung dieser Grundlagen vorgestellt, die für die Betrachtungen der nachfolgenden Kapitel benötigt werden: In 3.1 werden Prozesshierarchien und Stellvertreterprozesse eingeführt. Damit wird der Kontext beschrieben, in dem Anwendungsprogramme innerhalb einer Virtualisierungsumgebung arbeiten. Ziel dieser Betrachtungen ist es, Anwendungen zusammen mit ihrem jeweils unterlagerten Betriebssystem so zusammenzufassen, dass sie gegenüber einem Virtual Machine Monitor als einplanbare Einheit beschrieben werden können. In 3.2 wird ein Modell zur Abschätzung der mit der Virtualisierung einhergehenden Umschaltverluste vorgestellt. Dies ist erforderlich, um anschließend die praktische Brauchbarkeit der Planungsverfahren einzugrenzen, die in den meisten existierenden Virtual Machine Monitoren verwendet werden. In 3.3 wird eine Unterscheidung zwischen Ein- und Mehrprozessorbetriebssystemen eingeführt, und es wird begründet, weshalb ein Virtual Machine Monitor diese beiden Klassen von Gastsystemen auf unterschiedliche Weise zu behandeln hat.

3.1 Prozesshierarchien

In Abschnitt 2.3.6 wurde bereits kurz darauf eingegangen, wie ein Virtual Machine Monitor zwischen virtuellen Maschinen umschaltet. Innerhalb der virtuellen Maschinen arbeiten Gastbetriebssysteme, die ihrerseits zwischen einer Menge von Prozessen umschalten. Damit existieren in einer Virtualisierungsumgebung zwei Ebenen, auf denen Prozessumschaltungen stattfinden, d.h. es gibt eine *Hierarchie* von Prozessen. Im Folgenden wird ein Modell zur Beschreibung solcher Hierarchien vorgestellt.

Prozessumschaltungen werden durch das Ausführen bestimmter Befehlssequenzen implementiert, die den Zustand (z.B. die Registerinhalte) des bislang aktiven Prozesses im Speicher ablegen und den Zustand des zu aktivierenden Prozesses aus dem Speicher laden. Solche Sequenzen sind Teil des Schedulers[1], der, wie beschrieben, die Prozessorzuteilung zur Laufzeit vornimmt. Der gesamte Ausführungspfad aus Scheduler und der Gesamtheit aller Prozesse, zwischen denen er umschaltet, kann ebensogut auch als eine einzige Programmausführung aufgefasst werden, d.h. eine Menge von Prozessen kann, zusammen mit ihrem Scheduler, als ein einziger Prozess betrachtet werden, der die enthaltenen Prozesse vollkommen kapselt.

Die Eigenschaften dieses *Stellvertreterprozesses* sind abhängig von den Eigenschaften der gekapselten Prozesse und von dem enthaltenen *lokalen* Scheduler: Immer, wenn einer der enthaltenen Prozesse rechenwillig oder rechnend ist, gilt das auch für den Stellvertreterprozess. Darüber hinaus ist der Stellvertreterprozess auch dann rechnend, wenn Umschaltungen zwischen zwei gekapselten Prozessen stattfinden.

Damit ist jede Bereitzeit eines der gekapselten Prozesse zugleich auch eine Bereitzeit des Stellvertreterprozesses. Jede Startzeit ist größer oder gleich einer Startzeit des Stellvertreterprozesses, jeder Abschlusszeit des Stellvertreterprozesses geht mindestens eine Abschlusszeit eines gekapselten Prozesses voraus. Wenn beispielsweise $\sigma(t)$, wie in 2.2.8 beschrieben, ein zeitabhängiger Ausführungsplan für den lokalen Scheduler ist, so ist der Stellvertreterprozess immer dann rechnend, wenn einer der enthaltenen Prozesse rechnet, wenn also $\sigma(t) \neq 0$ ist.

Hieraus ergeben sich einige einfache Schlussfolgerungen:

- Die Prozessorauslastung durch einen Stellvertreterprozess entspricht der Summe der Auslastungen durch die gekapselten Prozesse, zuzüglich der Umschaltverluste, die im gekapselten (lokalen) Scheduler durch Prozesswechsel entstehen.
- Liegt für eine Menge von Prozessen ein zeitgesteuerter Ausführungsplan vor, so lässt sich auch für einen Stellvertreterprozess, der diese Prozessmenge kapselt, ein zeitgesteuerter Ausführungsplan angeben.
- Ist der lokale Scheduler eines Stellvertreterprozesses arbeitserhaltend (siehe 2.2.8) und ist unter den gekapselten Prozessen mindestens ein gieriger Prozess (siehe 2.2.2), so ist auch der Stellvertreterprozess gierig. Arbeitet der lokale Scheduler beispielsweise nach dem FIFO- oder dem Zeitscheibenverfahren (siehe 2.2.8), so ist der Stellvertreterprozess immer rechenbereit,

[1] Streng genommen sind sie eigentlich Teil des *Dispatchers*, vgl. 2.2.8

wenn unter allen darin gekapselten Prozessen mindestens einer rechenbereit ist.

- Ist umgekehrt unter den gekapselten Prozessen kein gieriger Prozess oder ist der lokale Scheduler eines Stellvertreterprozesse nicht arbeitserhaltend, so ist auch der Stellvertreterprozess nicht gierig, d.h. die durch ihn verursachte Prozessorauslastung ist beschränkt.

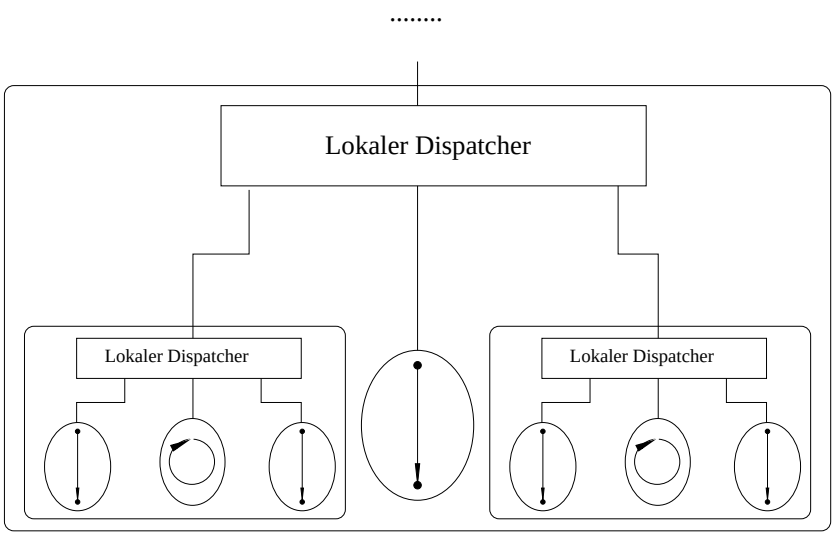

Abbildung 3.1: Beispiel einer Prozesshierarchie.

Ein weiterer, *globaler* Scheduler kann nun wiederum zwischen mehreren solcher Stellvertreterprozesse umschalten. Dabei ist es für diesen Scheduler nicht ersichtlich (und auch nicht relevant), ob es sich bei den ihm unterstellten Prozessen um einzelne Prozesse oder um Stellvertreter mehrerer Prozesse mit lokalem Scheduler handelt. Auf diese Weise lassen sich verschachtelte Hierarchien von Prozessen konstruieren (siehe Abbildung 3.1).

Die Schachtelungstiefe einer solchen Hierarchie kann prinzipiell beliebig groß werden. Für die Modellierung von Virtualisierungsumgebungen genügen jedoch Hierarchien mit nur wenigen[1] Ebenen: Auf der untersten Ebene gibt es den im Virtual Machine Monitor angesiedelten globalen Scheduler, der zwischen den als Stellvertreterprozesse zu beschreibenden virtuellen Maschinen umschaltet. Innerhalb der virtuellen Maschinen arbeiten Gastbetriebssysteme, die mit ihren lokalen Schedulern zwischen ihren jeweiligen Anwenderprozessen umschalten.

[1] An dieser Stelle werden zunächst nur zwei Hierarchieebenen begründet. Bei der Analyse in 4.1.5 wird sich jedoch erweisen, dass eine weitere Ebene zur Kapselung aller Nicht-Echtzeitgastsysteme erforderlich ist.

3.1.1 Beispiel: Zeitpartitionierung

Die Zeitpartitionierung ist ein Beispiel einer zweistufigen Prozesshierarchie. Es handelt sich um eine Methode, um voneinander unabhängigen Prozessen bzw. Prozessgruppen jeweils getrennte Mengen an Rechenzeit zuzuteilen, damit diese unabhängig voneinander operieren können. Die voneinander zu isolierenden Prozessgruppen werden, jeweils mit eigenen lokalen Schedulern als eine Einheit angesehen. Aus Sicht der Zeitpartitionierung ist das Verfahren, nach dem diese Prozessgruppen intern die ihnen zur Verfügung stehende Rechenzeit aufteilen, transparent. Aufgabe des Zeitpartitionierungs-Schedulers ist es, der Gesamtheit der Prozesse einer Gruppe Rechenkapazitäten zuzuweisen. Dieser globale Scheduler arbeitet ausschließlich zeitgesteuert (vgl. 2.2.8) nach einem periodischen Zeitplan. Abbildung 3.2 zeigt das Beispiel eines solchen Zeitplans, $\sigma_{part}(t)$. Analog gibt hier $\sigma_{part}(t) = k$, $k \in \{0, 1, 2, ...n\}$ die gesamte Prozessgruppe an, die zum gegebenen Zeitpunkt t aktiv ist. (Im Unterschied zur zeitgesteuerten Prozessorzuteilung, wie sie in 2.2.8 vorgestellt wurde, ist hier immer eine der Prozessgruppen aktiv, d.h. es gibt keinen Wert k von $\sigma_{part}(t)$, für den keine Prozessgruppe aktiv wäre.) Wie das Beispiel von Abbildung 3.2 zeigt, kann eine Prozessgruppe innerhalb einer Periode auch mehrfach rechnend sein.

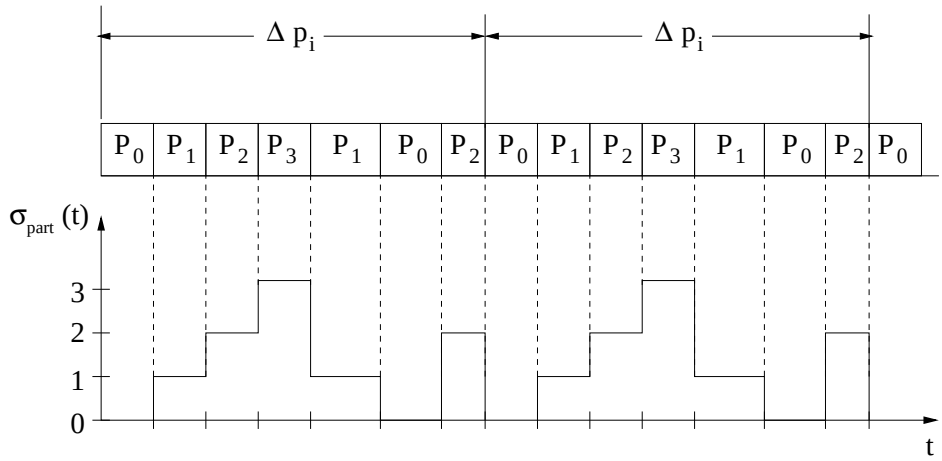

Abbildung 3.2: Beispiel eines Zeitplanes bei Zeitpartitionierung.

Jede Prozessgruppe wird dadurch zyklisch wiederholt für eine feste Zeitdauer ausgeführt. Damit verfügt jede von ihnen über ein fest zugesichertes Kontingent an Rechenzeit, das ihr lokaler Scheduler nach jeweils eigenen Regeln an seine untergeordneten Prozesse weitergeben kann. Die Prozessgruppen arbeiten aus der Sicht des globalen Schedulers als periodische Prozesse. Da sie jeweils über einen festen Anteil an der Periodendauer des Systems verfügen, werden sie in diesem Zusammenhang auch als *Zeitpartitionen* bezeichnet. Die Prozessgruppen müssen stets

in der Lage sein, die Rechenkapazität, die der globale Scheduler ihnen zuweist, vollständig zu „konsumieren". Jede von ihnen enthält deshalb immer einen gierigen „idle-"Prozess, der, falls kein anderer Prozess rechenbereit ist, überschüssige Rechenkapazität verbraucht.

Da der globale Scheduler ausschließlich zeitgesteuert arbeitet, sind für alle ihm unterstellten Prozessgruppen die Zeitpunkte und Dauern ihrer Aktivierung exakt vorherbestimmt. Daher können Programme, die in einer Zeitpartition arbeiten, ohne Kenntnis der Vorgänge in anderen Zeitpartitionen eigenständig entscheiden, ob sie eine Echtzeitaufgabe übernehmen können. Falls es dabei zu einer Fehlentscheidung kommt, d.h. falls eine zugesagte Frist nicht eingehalten wird, so wirkt sich dies ausschließlich innerhalb der Zeitpartition aus, in der die fehlerhafte Zusage entschieden wurde, während andere Zeitpartitionen nicht beeinflusst werden. Somit können mit dieser Methode Subsysteme zuverlässig voneinander zeitlich entkoppelt werden.

Nachteilig ist allerdings, dass durch die starre Zeitzuteilung jede Zeitpartition gezwungen ist, die ihr zugewiesene Zeit vollständig zu verbrauchen: Die zugeteilten Rechenkapazitäten müssen für den denkbar ungünstigsten Fall bemessen werden. Dieser ungünstigste Fall tritt aber in der Realität nur sehr selten auf, weshalb ein erheblicher Teil der Rechenkapazität in der Regel nicht genutzt werden kann. Daher führt die Zeitpartitionierung zu der für Echtzeitsysteme typischen, geringen Prozessorauslastung.

Die Zeitpartitionierung wird heute vor allem im Bereich der Avionik angewendet. Zum Beispiel wird in [ARI97] eine standardisierte Programmierschnittstelle für Avioniksysteme beschrieben, bei der die Prozessorzuteilung nach diesem Verfahren vorgeht.

3.1.2 Beispiel: Prozessausführung in virtuellen Maschinen

Das in 2.2.1 eingeführte Prozessmodell kann unmittelbar auf virtuelle Maschinen angewendet werden: ein Virtual Machine Monitor ist ein Prozesstyp (hier mit V bezeichnet), und virtuelle Maschinen sind verschiedene Ausführungen des Prozessobjektes V_i. Die unterschiedlichen Programme, die von den virtuellen Maschinen ausgeführt werden, stellen dabei die Daten dar, hinsichtlich derer sich diese Ausführungen voneinander unterscheiden. Nach dem Ende einer Prozessausführung[1] kann diese erneut gestartet werden. Somit kann eine virtuelle Maschine als eine Reihe zeitdiskreter Ausführungen des Prozessobjektes, V_i^j beschrieben

[1]Zum Beispiel, wenn die virtuelle Maschine bei der Befehlsabarbeitung auf einen `HALT`-Maschinenbefehl stößt.

werden. Umschaltungen zwischen diesen Prozessausführungen werden von einem globalen Scheduler vorgenommen, der meist ein Teil des Programmcodes des Virtual Machine Monitor ist. Dieser Scheduler wird im Weiteren als *VM-Scheduler* bezeichnet.

Die Bereitzeiten und Fristen der Prozessausführungen werden von der Anwendung, hier also von Programmen, die von den virtuellen Maschinen ausgeführt werden, vorgegeben. Ebenso wie die Zeitpartitionierung ist die Prozessausführung in virtuellen Maschinen ein Beispiel für eine 2-stufige Prozesshierarchie: Auf der Ebene des Virtual Machine Monitors schaltet der globale VM-Scheduler zwischen den virtuellen Maschinen um. Innerhalb der virtuellen Maschinen schalten die lokalen Scheduler der Gastbetriebssysteme zwischen den Prozessen der Gastsysteme um (siehe Abbildung 3.3).

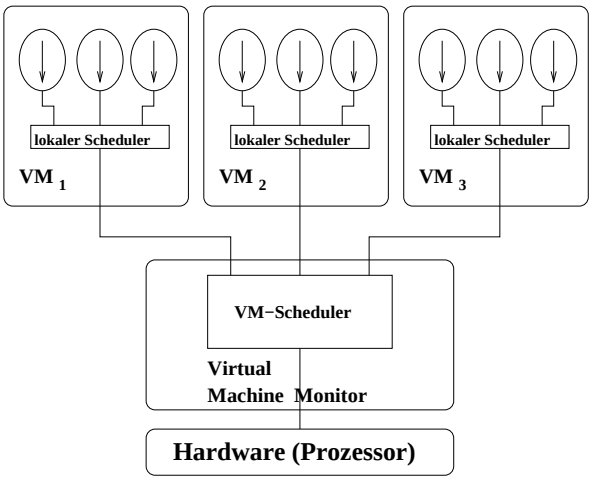

Abbildung 3.3: Virtual Machine Monitor mit Gastsystemen.

Das Zeitverhalten eines Prozesses, der unter der Verwaltung eines Gastbetriebssystems steht, ist damit nicht allein vom Scheduler dieses Gastsystems, sondern auch vom globalen VM-Scheduler abhängig. Der VM-Scheduler hat die Aufgabe, die verfügbare Rechenzeit anteilig über mehrere solcher Gastsysteme zu verteilen. Eine einfache Möglichkeit hierzu ist das in 2.2.8 beschriebene Zeitscheibenverfahren: Jedem Gastsystem wird der Prozessor zyklisch für die Dauer seiner jeweiligen Zeitscheibe zugeteilt. Damit sind die virtuellen Maschinen zunächst periodische Prozesse, d.h. es gelten die gleichen Verhältnisse wie bei der Zeitpartitionierung: Die Rechenzeiten Δe_i sind die Zeitscheibendauern. Die Periodendauer Δp_i ist für alle virtuellen Maschinen gleich. Sie ist durch die Summe der Zeitscheibendauern gegeben. Abbildung 3.4 zeigt die zeitlichen Zusammenhänge und den sich ergebenden Ausführungsplan des VM-Schedulers $\sigma_{vm}(t)$.

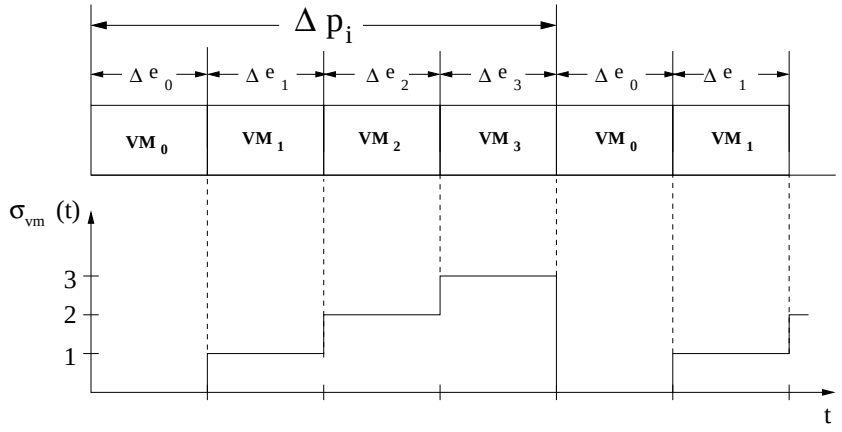

Abbildung 3.4: Virtuelle Maschinen als periodische Prozesse.

Bei dieser Vorgehensweise kann es vorkommen, dass einer virtuellen Maschine vom VM-Scheduler Rechenkapazität zugewiesen wird, während das in ihr arbeitende Gastsystem gerade keinen Bedarf für Rechenkapazität hat. Bei der zuvor beschriebenen Zeitpartitionierung wurde für diesen Fall gefordert, dass das Gastsystem die überschüssige Rechenkapazität verbraucht, indem es seinen internen „idle-"Prozess ausführt. Bei der Virtualisierung wird ein effizienter Umgang mit Betriebsmitteln höher bewertet als zeitlich deterministisches Verhalten. Deshalb kann hier diese Verschwendung von Rechenkapazitäten nicht akzeptiert werden. Stattdessen sollte das betroffene Gastsystem den Rest seiner Zeitscheibe aufgeben, sodass der Prozessor unmittelbar an das nächste Gastsystem weitergegeben werden kann.

Technisch kann dies bei einer Paravirtualisierungs-Umgebung relativ einfach erreicht werden, indem der „idle-"Prozess des Gastsystems entsprechend modifiziert wird: Statt der sonst üblichen Endlosschleife wird der VM-Scheduler aufgerufen, der daraufhin zur nächsten virtuellen Maschine umschaltet. Bei einer Vollvirtualisierung besteht diese Möglichkeit zunächst nicht, da Anpassungen an Gastsystemen in der Regel nicht möglich sind. Dennoch kann auch hier das gleiche Ergebnis erreicht werden, wenn das Gastsystem in seinem „idle-"Prozess einen speziellen Maschinenbefehl verwendet, mit dem der Prozessor in einen energiesparenden Modus gebracht wird. Bei den meisten heutigen Betriebssysteme ist das der Fall (vgl. 2.2.2), und der betreffende Maschinenbefehl ist privilegiert, sodass seine Ausführung in einer nicht-privilegierten virtuellen Maschine zum Aufruf des Virtual Machine Monitors führt. Dieser kann dann wiederum den VM-Scheduler aufrufen.

Durch die vorzeitige Freigabe des Prozessors wird die bei der Zeitpartitionierung festgestellte schlechte Prozessorauslastung vermieden, allerdings sind dadurch die

virtuellen Maschinen keine periodischen Prozesse mehr. Der Ausführungsplan des VM-Schedulers ist keine reine Zeitfunktion, d.h. die Prozessorzuteilung auf dieser Ebene erfolgt nun dynamisch, nicht statisch.

Die Umschaltzeitpunkte der virtuellen Maschinen hängen damit auch vom Verhalten der vorangegangenen virtuellen Maschinen ab. Ein in einer virtuellen Maschine arbeitender Prozess kann nicht vorherbestimmen, ob er zu einem in der Zukunft liegenden Zeitpunkt einen Prozessor zur Verfügung haben wird. Somit eignen sich virtuelle Maschinen, die nicht rein zeitgesteuert aktiviert werden, nur bedingt als Laufzeitumgebung für Echtzeitprozesse.

3.1.3 Kapselung von Echtzeitprozessen

Es stellt sich die Frage, wie, bzw. durch welche Prozessparameter ein Stellvertreterprozess, der eine Menge von Echtzeitprozessen kapselt, gegenüber einem globalen Scheduler dargestellt werden kann. Es wurde bereits festgestellt, dass ein solcher Stellvertreterprozess nicht gierig ist, dass die durch ihn verursachte Auslastung also beschränkt ist. Diese Auslastung ist die Summe der Auslastungen durch die gekapselten Prozesse zuzüglich der Auslastung durch die Ausführung des lokalen Schedulers. Für die Planung ist neben dieser Aussage über die Auslastung, also den *Bedarf* an Rechenkapazität, noch eine Angabe über die Zeitpunkte und -intervalle erforderlich, in denen diese Rechenkapazitäten vom globalen Scheduler zur Verfügung gestellt werden müssen, damit ein lokaler Scheduler seine Prozesse korrekt einplanen kann.

Für die folgenden Überlegungen wird von unterbrechbaren periodischen Prozessen ausgegangen (siehe 2.2.1). Nahezu alle im Rahmen der praktischen Echtzeitverarbeitung anstehenden Aufgabenstellungen lassen sich auf dieses Prozessmodell abbilden. Es existiert eine Reihe von Verfahren zur Erstellung von Ausführungsplänen für eine Menge unterbrechbarer periodischer Prozesse, von denen mit RMS und EDF in 2.2.7 die beiden meist-verwendeten (weil praktikablen) Verfahren eingeführt wurden. Im Folgenden wird für diese beiden Verfahren untersucht, ob und wie sie in einer Prozesshierarchie durch einen lokalen Scheduler anwendbar sind. Bei beiden Überlegungen wird von einem Modell von „Störprozessen" ausgegangen, das im Folgenden zunächst vorgestellt wird:

Störprozess-Modell

Es wird das Modell einer 2-stufigen Prozesshierarchie zugrunde gelegt, wobei der globale Scheduler zunächst rein zeitgesteuert periodisch mit einer noch zu bestimmenden Periodendauer Δp_{glob} arbeitet. Unter einem solchen Scheduler steht

einem Stellvertreterprozess für die Gesamtheit seiner Prozesse ein periodisch wiederholtes Zeitfenster zur Verfügung (siehe Abbildung 3.5).

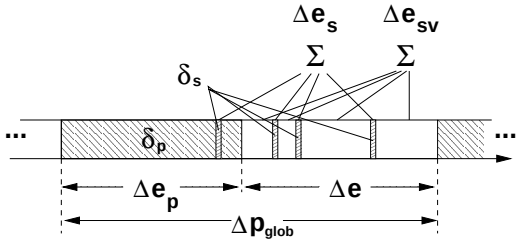

Abbildung 3.5: Zeitfenster und Störprozesse δ_s und δ_p.

Mit Δe als Dauer dieses Zeitfensters und Δp_{glob} als Periodendauer des globalen Schedulers lässt sich dieser Sachverhalt aus Sicht des lokalen Schedulers mit der Annahme eines nicht unterbrechbaren, periodischen „Störprozesses" δ_p beschreiben, der während des restlichen Teils des Zyklus, also dem Teil der nicht dem betrachteten Stellvertreterprozess zugewiesen ist, aktiv ist. Für seine Ausführungszeit Δe_p und seine Periodendauer Δp_p gilt:

$$\Delta e_p = \Delta p_{glob} - \Delta e, \Delta p_p = \Delta p_{glob}$$

Anmerkung: Bis zu diesem Punkt entspricht das Modell der „slotted priority" Architektur von Bollella [Bol97]. Der im Folgenden zusätzlich betrachtete, optionale sporadische Störprozess kommt dort hingegen nicht vor.

Da der Stellvertreterprozess als unterbrechbar angesehen wird, kann er ggf. auch während seines Zeitfensters unterbrochen werden. Dazu wird ein weiterer, sporadischer Störprozess δ_s eingeführt, dessen kumulierte Rechenzeit Δe_s innerhalb einer Periode des globalen Schedulers beschränkt (und bekannt) sei. Diese Rechenzeit kann im ungünstigsten Fall vollständig im Zeitfenster des Stellvertreterprozesses liegen (d.h. Δe wird maximal um Δe_s geschmälert). Die Periodendauer dieses Störprozesses, Δp_s ist ebenfalls gleich der Periodendauer Δp_{glob} des globalen Schedulers.

Um zu entscheiden, ob eine durch einen Stellvertreterprozess gekapselte Prozessmenge innerhalb einer Prozesshierarchie planbar ist, werden die beiden Störprozesse δ_p und δ_s zur Prozessmenge hinzugefügt. Dabei ist zu beachten, dass diese Störprozesse faktisch Vorrang vor dem Stellvertreterprozess haben: Immer wenn einer von ihnen rechenwillig ist, entzieht der globale Scheduler dem Stellvertreterprozess den Prozessor. Die zur Prozessmenge hinzuzufügenden Störprozesse müssen also gegenüber den gekapselten Prozessen als höher priorisiert angesehen

werden. Im Verhältnis zu den gekapselten Prozessen müssen die Eigenschaften der hinzukommenden Störprozesse δ_p und δ_s daher so sein, dass der lokale Scheduler bei Anwendung seines Planungsverfahrens sie exakt so einplanen würde, wie der globale Scheduler sie (ohne Kenntnis des lokalen Schedulers) tatsächlich einplant. Ist dies der Fall, so kann die neue, um die Störprozesse erweiterte Prozessmenge nach dem bestehenden Verfahren eingeplant werden. Welche Einschränkungen sich daraus jeweils ergeben, hängt –neben den Eigenschaften der Prozessmenge und der Störprozesse– auch vom Planungsverfahren des lokalen Schedulers ab. Im Folgenden werden diese Überlegungen für die Planungsverfahren RMS und EDF vollzogen:

Kapselung von nach RMS geplanten Prozessen

Um RMS (siehe 2.2.7) auch als lokalen Scheduler innerhalb eines Stellvertreterprozesses anwenden zu können, müssen die beiden Störprozesse δ_p und δ_s mitberücksichtigt werden. Beide haben die gleiche Periodendauer Δp_{glob}. Sie sind aus Sicht des Stellvertreterprozesses nicht unterbrechbar, was zunächst gegen eine Anwendung des Verfahrens zu sprechen scheint. Allerdings fordert RMS genau genommen Unterbrechbarkeit nur von denjenigen Prozessen, deren Periodendauern größer als die kleinste aller betrachteten Periodendauern sind.

Sofern also die Periodendauern aller einzuplanenden Prozesse größer oder gleich der globalen Periodendauer sind, können die beiden Störprozesse als höchstpriorisierte Prozesse des Systems eingeplant werden.

Formal ausgedrückt: Gegeben die unterbrechbaren Prozesse $P = \{1, ..., n\}$ mit den Periodendauern Δp_i, $i = 1...n$, die nach RMS-Vorschrift priorisiert sind (vgl. Gleichung (2.5) in Abschnitt 2.2.7)

Wird diese Prozessgruppe in einem Stellvertreterprozess gekapselt, so wird dies modelliert, indem ihr die beiden Störprozesse δ_s und δ_p hinzugefügt werden. Falls die kürzeste aller Periodendauern der Prozesse aus P, Δp_1, größer oder gleich der Periodendauer der beiden Störprozesse ist:

$$\Delta p_1 \geq \Delta p_{glob} \tag{3.1}$$

so ist auch die resultierende Prozessmenge $P' = \{\delta_p, \delta_s, 1, ..n\}$ nach RMS-Vorschrift (2.5) priorisiert.

Die nun folgenden Überlegungen sind an [SL03] angelehnt. In der genannten Arbeit wird die Anwendbarkeit des RMS-Verfahrens auf Prozesse untersucht, die innerhalb einer periodischen Zeitscheibe arbeiten. Darüber hinausgehend werden

im Folgenden zusätzlich auch sporadische Prozesse berücksichtigt, die in der Lage sind, die periodische Zeitscheibe zu unterbrechen.

Ohne Verlust der Allgemeinheit wird eine nach RMS-Vorschrift (2.5) priorisierte Prozessmenge $Q = \{1, ..., m\}$ betrachtet, wobei es sich bei den ersten beiden Prozessen, 1 und 2 , um die erwähnten Störprozesse δ_s und δ_p handelt, deren Periodendauern und Ausführungszeiten konstant und bekannt seien.

In [LL73] zeigen Liu und Layland, dass für einen unabhängigen periodischen Prozess die Abschlusszeit maximal wird, wenn:

1. alle höher priorisierten Prozesse dieselbe Bereitzeit haben wie der betrachtete Prozess,
2. das Verhältnis der Periodendauern zweier beliebiger aufeinanderfolgender Prozesse kleiner als 2 ist, also:

$$\forall i, i = 1...m-1 : \frac{\Delta p_{i+1}}{\Delta p_i} < 2 \tag{3.2}$$

3. die Ausführungszeit jedes Prozesses, Δe_i, gleich der Differenz der Periodendauer dieses Prozesses und der seines Nachfolgers ist, also:

$$\forall i, i = 1...m-1 : \Delta e_i = \Delta p_{i+1} - \Delta p_i \tag{3.3}$$

Unter diesen Voraussetzungen gilt für die Ausführungszeit des niedrigst-priorisierten Prozesses (siehe [LL73]):

$$\Delta e_m = \Delta p_m - 2\sum_{i=1}^{m-1} \Delta e_i$$

$$= \Delta p_m - 2(\Delta p_2 - \Delta p_1 + \Delta p_3 - \Delta p_2.... + \Delta p_m - \Delta p_{m-1})$$

$$\Delta e_m = 2\Delta p_1 - \Delta p_m \tag{3.4}$$

Wir suchen nun nach einer Untergrenze, d.h. einem Minimum der Auslastung, für die eine Planbarkeit der betrachteten Prozessgruppe unter ungünstigsten Annahmen noch gegeben ist. Die Gesamtauslastung des Prozessors durch die Prozessgruppe ist:

$$U = \sum_{i=1}^{m} \frac{\Delta e_i}{\Delta p_i}$$

Mit (3.3) wird daraus:

$$U = \frac{\Delta e_m}{\Delta p_m} + \sum_{i=1}^{m-1} \frac{\Delta p_{i+1} - \Delta p_i}{\Delta p_i}$$

Mit (3.4):

$$U = \frac{2\Delta p_1 - \Delta p_m}{\Delta p_m} + \sum_{i=1}^{m-1} \frac{\Delta p_{i+1} - \Delta p_i}{\Delta p_i}$$
$$U = \frac{2\Delta p_1}{\Delta p_m} - m + \sum_{i=1}^{m-1} \frac{\Delta p_{i+1}}{\Delta p_i}$$

Wir definieren $R_i = \frac{\Delta p_{i+1}}{\Delta p_i}, \forall i : i = 1...m-1$.
Man beachte, dass: $\frac{\Delta p_m}{\Delta p_1} = \frac{\Delta p_2}{\Delta p_1} \cdot \frac{\Delta p_3}{\Delta p_2} .... \frac{\Delta p_m}{\Delta p_{m-1}} = R_1 \cdot R_2 \cdot R_3 .... R_{m-1}$. Damit ergibt sich:

$$U = 2 \prod_{i=1}^{m-1} [\frac{1}{R_i}] - m + \sum_{i=1}^{m-1} R_i \tag{3.5}$$

Die Auslastungsbeiträge der beiden Störprozesse δ_p und δ_s sind:

$$U_p = \frac{\Delta e_p}{\Delta p_p} = \frac{\Delta e_p}{\Delta p_{glob}}$$
$$U_s = \frac{\Delta e_s}{\Delta p_s} = \frac{\Delta e_s}{\Delta p_{glob}}$$

Damit ergibt sich[1]:

$$R_1 = R_p = \frac{\Delta p_2}{\Delta p_1} = \frac{\Delta p_{glob}}{\Delta p_{glob}} = 1 \tag{3.6}$$

[1]Da beide Störprozesse die gleiche Periodendauer (Δp_{glob}) besitzen, ist ihre Reihenfolge gleichgültig. Wir wählen hier willkürlich δ_p als ersten und δ_s als zweiten Prozess

$$R_2 = R_s = \frac{\Delta p_3}{\Delta p_2} = \frac{\Delta e_p + \Delta e_s + \Delta p_{glob}}{\Delta p_{glob}} = \frac{\Delta e_p}{\Delta p_{glob}} + \frac{\Delta e_s}{\Delta p_{glob}} + 1 = U_p + U_s + 1 \tag{3.7}$$

Damit wird aus (3.5):

$$U = \frac{2}{R_s R_p} \prod_{i=3}^{m-1} [\frac{1}{R_i}] - m + R_s + R_p + \sum_{i=3}^{m-1} R_i$$

Wir kehren nun wieder zu unserer alten, auf die Nutzerprozessgruppe $P = \{1, ..., n\}$ bezogenen Indizierung zurück. Dabei gilt: $m = n + 2$:

$$U = \frac{2}{R_s R_p} \prod_{i=1}^{n-1} [\frac{1}{R_i}] - (n+2) + R_s + R_p + \sum_{i=1}^{n-1} R_i \tag{3.8}$$

Hierin sind R_s und R_p konstant. Wir suchen nun ein Minimum für die Auslastung in Abhängigkeit der Faktoren R_i. Nullsetzen der Ableitungen $\frac{dU}{dR_j}, \forall j : j = 1...n-1$ liefert n Gleichungen der Form:

$$\frac{dU}{dR_j} = 1 - \frac{2}{R_s R_p} \frac{1}{R_j} \prod_{i=1}^{n-1} [\frac{1}{R_i}] = 0$$

$$R_s \cdot R_p \cdot R_j \prod_{i=1}^{n-1} R_i = 2 \tag{3.9}$$

Durch Division jeweils zweier beliebiger dieser Gleichungen (z.B. der k-ten und der l-ten) ergibt sich:

$$\forall k, l : k, l = 1...n : R_k = R_l$$

D.h. ein Minimum für die Auslastung wird erreicht, wenn alle R_i gleich sind, d.h. $R_i = R, \forall i : i = 1...n$. Dies eingesetzt in (3.9) liefert:

$$R^n = \frac{2}{R_s R_p}$$

$$R = \sqrt[n]{\frac{2}{R_s R_p}}$$

Dies eingesetzt in (3.8) ergibt die minimale Auslastung zu:

$$\begin{aligned} U_{min} &= \frac{2}{R_s R_p} \cdot \frac{1}{[(\frac{2}{R_s R_p})^{\frac{1}{n}}]^{n-1}} - (n+2) + R_s + R_p + (n-1) \cdot (\frac{2}{R_s R_p})^{\frac{1}{n}} \\ &= n \cdot (\frac{2}{R_s R_p})^{\frac{1}{n}} - n - 2 + R_s + R_p \\ &= n \cdot ((\frac{2}{R_s R_p})^{\frac{1}{n}} - 1) + R_s + R_p - 1 \end{aligned}$$

Mit (3.6) und (3.7):

$$U_{min} = n \cdot ((\frac{2}{U_s + U_p + 1})^{\frac{1}{n}} - 1) + U_s + U_p \tag{3.10}$$

Hierin sind U_s und U_p die konstanten Auslastungsbeiträge der Störprozesse δ_s und δ_p. Die Auslastung durch die Prozessmenge P, die unter dem lokalen Scheduler arbeiten, ist somit:

$$U^P_{min} = n \cdot ((\frac{2}{U_s + U_p + 1})^{\frac{1}{n}} - 1) \tag{3.11}$$

Für große Anzahlen n von Prozessen wird:

$$\lim_{n \to \infty} U^P_{min} = \ln(\frac{2}{U_s + U_p + 1}) \tag{3.12}$$

Abbildung 3.6 zeigt den Verlauf dieser Funktion. Man beachte, dass für $U_s = U_p = 0$, d.h. ohne Störprozesse, der bekannte Grenzwert $\ln(2)$ angenommen wird. Für $U_s + U_p = 1$ (d.h. eine vollständige Systemauslastung durch die Störprozesse) ergibt sich $\ln(1) = 0$, was unmittelbar einzusehen ist. Wir haben also, wie bei der klassischen Planung nach monotonen Raten, ein einfach anzuwendendes Kriterium für die Planbarkeit von Prozessmengen, die durch einen Stellvertreterprozess gekapselt werden.

Bei naiver Betrachtungsweise könnte man erwarten, dass bei einem Stellvertreterprozess, der einen bestimmten Anteil an der gesamten Periodendauer besitzt, der Wert der Auslastung, für die das RMS-Verfahren sicher einen brauchbaren Plan liefert, einfach proportional zu diesem Anteil skaliert werden kann. Tatsächlich aber verläuft die Funktion nur annähernd linear, wie der Vergleich mit der punktierten Linie in Abbildung 3.6 zeigt. Die erreichbare Auslastung liegt um bis zu 6% unter dem entsprechend skalierten Wert.

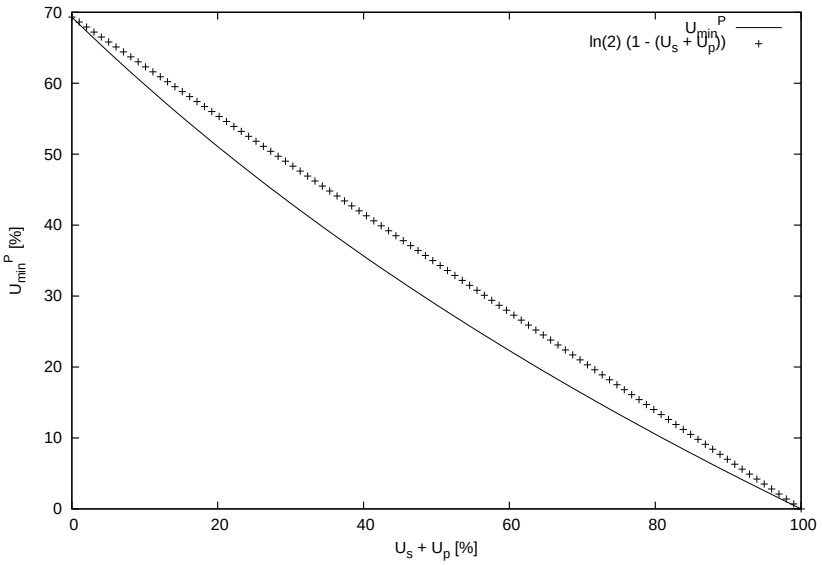

Abbildung 3.6: Planbarkeitskriterium für RMS in lokalen Schedulern.

Dennoch ist die Anwendung des RMS-Verfahrens innerhalb eines Stellvertreterprozesses grundsätzlich gesichert, sofern die kürzeste Periodendauer unter allen gekapselten Prozessen größer oder gleich der Periodendauer des globalen Schedulers ist, und solange die Gesamtauslastung durch die Prozesse nicht über dem nach (3.10) bzw. (3.11) ermittelten Grenzwert liegt.

Diese Erkenntnis lässt sich auch auf den Standpunkt des globalen Schedulers beziehen: Der Stellvertreterprozess kann gegenüber diesem als Ganzes durch einen periodischen Prozess beschrieben werden, dessen Periodendauer gleich der kleinsten Periodendauer unter allen gekapselten Prozessen ist. Formal: Gegeben die unterbrechbare, nach RMS-Vorschrift priorisierte (s.o.) Prozessmenge $P = \{1, ..., n\}$ mit den Periodendauern Δp_i, $i = 1...n$. Diese Prozessmenge kann durch einen einzigen Stellvertreterprozess mit der Periodendauer

$$\Delta p_{sv} = min(\{\Delta p_i | i \in P\}) = \Delta p_1 \tag{3.13}$$

substituiert werden.

Damit der Stellvertreterprozess dabei alle seine gekapselten Prozesse fristgerecht ausführen kann, muss der globale Scheduler ihm jeweils innerhalb einer Periode ein Zeitfenster zuweisen, dessen Dauer Δe_{sv} durch den Gesamtbedarf der gekapselten Prozesse bestimmt ist. Aufgrund der Verwendung des RMS-Verfahrens kommt es innerhalb dieses Zeitfensters zu Planungslücken, während derer keiner der gekapselten Prozesse ausgeführt werden kann. Diese sind in der Dauer des

Zeitfensters zu berücksichtigen, d.h. der globale Scheduler muss vorab mehr als die Summe der Auslastungen durch die gekapselten Prozesse zuweisen, damit eine hinreichende Reserve besteht. Gleichung (3.11) gibt eine Beziehung zwischen der zuzuweisenden Rechenkapazität und der im ungünstigsten Fall tatsächlich genutzten Rechenkapazität an.

Im Störprozess-Modell (s.o.) ist die Dauer des Zeitfensters Δe_{sv} des Stellvertreterprozesses durch den nicht den Störprozessen zugewiesenen Anteil der Zykluszeit gegeben:

$$U_{sv} = \frac{\Delta e_{sv}}{\Delta p_{sv}} = 1 - U_p - U_s$$

Die Gesamtauslastung durch die Prozessmenge P ist:

$$U(P) = \sum_{i=1}^{n} \frac{\Delta e_i}{\Delta p_i}$$

Dies in Gleichung (3.11) eingesetzt und nach U_{sv} aufgelöst ergibt:

$$U_{sv} = 2 - \frac{2}{\left(\frac{U(P)}{n} + 1\right)^n} \tag{3.14}$$

$$\Delta e_{sv} = 2 \cdot \Delta p_{sv} \cdot \left(1 - \frac{1}{\left(\frac{U(P)}{n} + 1\right)^n}\right) \tag{3.15}$$

Bzw. für große Anzahlen n von Prozessen:

$$U_{sv} = 2 \cdot \left(1 - e^{-U(P)}\right) \tag{3.16}$$

$$\Delta e_{sv} = 2 \cdot \Delta p_{sv} \cdot \left(1 - e^{-U(P)}\right) \tag{3.17}$$

Mit Gleichungen (3.15) und (3.17) stehen somit einfache Beziehungen zur Verfügung, die es gestatten, für eine gegebene, nach RMS priorisierte Prozessmenge mit der kleinsten Periodendauer Δp_{sv} und der Auslastung $U(P)$ die Prozessparameter Periodendauer und Auslastung (bzw. Ausführungszeit) eines periodischen unterbrechbaren Stellvertreterprozesses zu ermitteln, der aus der Sicht eines globalen Schedulers zu dieser Prozessmenge äquivalent ist. Der globale Scheduler kann nun seinerseits mehrere solcher Stellvertreterprozesse nach einem eigenen Verfahren –beispielsweise RMS oder EDF– planen.

Kapselung von nach EDF geplanten Prozessen

Die folgenden Überlegungen sind an [Bol97] angelehnt. Im Unterschied zu der genannten Arbeit wird hier wiederum zusätzlich zu dem periodischen Störprozess δ_p auch die Störung durch einen sporadischen Störprozess δ_s berücksichtigt.

Betrachtet wird eine Menge unterbrechbarer Prozesse $P = \{1, ..., n\}$ mit den Periodendauern Δp_i, $i = 1...n$, die nach EDF-Vorschrift geplant werden, d.h. ausgeführt wird jeweils derjenige Prozess, dessen Frist am kürzesten ist.

Diese Prozesse werden nun durch einen Stellvertreterprozess gekapselt, der in einem periodisch wiederholten Zeitfenster ausgeführt wird. Die Periodendauer dieses Zeitfensters sei Δp_{sv}, seine Dauer sei Δe_{sv}. Die restliche Zeit der Periode sei den Störprozessen δ_p und δ_s zugewiesen, d.h. deren Periodendauern seien ebenfalls gleich Δp_{sv}, und ihre Ausführungszeiten seien Δe_s und Δe_p (siehe Abbildung 3.5).

Zunächst soll für ein beliebiges Zeitintervall $[t, t + \Delta t]$ die Menge an Rechenzeit ermittelt werden, die dem Stellvertreterprozess mindestens zugeteilt wird. Die Anzahl ganzer Zeitscheiben während des Zeitintervalls ist:

$$N = \left\lfloor \frac{\Delta t}{\Delta p_{sv}} \right\rfloor$$

Während des Zeitintervalls $[t, t + \Delta t]$ werden dem Stellvertreterprozess mindestens N Zeitscheiben der Länge Δe_{sv} zugeteilt, während den Störprozessen ebenfalls N Zeitscheiben der Längen Δe_s bzw. Δe_p zugeteilt werden. Von der übrig bleibenden Zeit:

$$\begin{aligned} \Delta t - N \cdot (\Delta e_{sv} + \Delta e_s + \Delta e_p) &= \Delta t - N \cdot \Delta p_{sv} \\ &= \Delta t - \left\lfloor \frac{\Delta t}{\Delta p_{sv}} \right\rfloor \cdot \Delta p_{sv} = \Delta t \bmod \Delta p_{sv} \end{aligned}$$

können maximal $\Delta e_s + \Delta e_p$ den Störprozessen zugewiesen werden. Ist also $\Delta e_s + \Delta e_p \leq \Delta t \bmod \Delta p_{sv}$, so erhält der Stellvertreterprozess einen Anteil von $\Delta t \bmod \Delta p_{sv} - \Delta e_s - \Delta e_p$. Damit ist die dem Stellvertreterprozess innerhalb des Intervalls $[t, t + \Delta t]$ mindestens zugewiesene Rechenzeit:

$$\left\lfloor \frac{\Delta t}{\Delta p_{sv}} \right\rfloor \cdot \Delta e_{sv} + \max(0, \Delta t \bmod \Delta p_{sv} - \Delta e_s - \Delta e_p) \qquad (3.18)$$

Aufbauend auf diesen Überlegungen soll nun gezeigt werden, dass eine Menge unterbrechbarer Prozesse $P = \{1, ..., n\}$ mit den Periodendauern Δp_i und den Ausführungszeiten Δe_i, $i = 1...n$, die durch einen lokalen Scheduler nach EDF-Vorschrift eingeplant werden, unter einem globalen Scheduler mit der Periodendauer Δp_{sv}, von der die Ausführungszeit Δe_{sv} der Prozessmenge zugeteilt wird, während die Ausführungszeiten Δe_s und Δe_p anderen Prozessen bzw. Prozessmengen zugeteilt werden[1], genau dann brauchbar geplant werden, wenn für alle $t \geq 0$ gilt:

$$\sum_{i=1}^{n} \left\lfloor \frac{t}{\Delta p_i} \right\rfloor \Delta e_i \leq \left\lfloor \frac{t}{\Delta p_{sv}} \right\rfloor \cdot \Delta e_{sv} + \max(0, t \bmod \Delta p_{sv} - \Delta e_s - \Delta e_p) \quad (3.19)$$

Dass Ungleichung (3.19) ein notwendiges Kriterium für die Planbarkeit darstellt, kann folgendermaßen begründet werden:

Während eines Zeitintervalls $[0, t]$ benötigt jeder der Prozesse $\left\lfloor \frac{t}{p_i} \right\rfloor \cdot \Delta e_i$ Zeiteinheiten. Die linke Seite von Ungleichung (3.19) stellt also den Bedarf an Prozessorzeit des Prozessmenge P im Intervall $[0, t]$ dar. Nach (3.18) stellt die rechte Seite von Ungleichung (3.19) die Prozessorzeit dar, die der Prozessmenge im Intervall $[0, t]$ mindestens zugeteilt wird. Diese muss offensichtlich mindestens dem Bedarf der Prozessmenge entsprechen, damit die Auslastung des Prozessors 100% nicht übersteigt.

Es bleibt die Notwendigkeit von Ungleichung (3.19) zu zeigen. Dazu wird angenommen, dass für eine gegebene Prozessmenge P Ungleichung (3.19) $\forall t, t >= 0$ erfüllt sei, und dennoch komme es zu einer Fristüberschreitung zum Zeitpunkt t_d. Sei $t_1 \leq t_d$ entweder:

- Der Endzeitpunkt des letzten der Prozessmenge P zugeteilten Zeitintervalls, während dessen keiner der Prozesse $\in P$ ausgeführt wurde, oder
- Der letzte Zeitpunkt vor t_d, zu dem ein Prozess $\in P$ mit einer Frist später als t_d ausgeführt wurde,

je nachdem, welcher der beiden Zeitpunkte der spätere ist.

Aufgrund dieser Wahl von t_1 kann während des Intervalls $[t_1, t_d]$ kein Prozess mit einer Frist nach t_d ausgeführt werden. Da die Prozessmenge nach EDF-Vorschrift geplant ist, ist der Rechenzeitbedarf innerhalb des Zeitintervalls $[t_1, t_d]$ gleich

[1] D.h. $\Delta e_{sv} + \Delta e_p + \Delta e_s = \Delta p_{sv}$

$\sum_{i=1}^{n} \left\lfloor \frac{t_d - t_1}{\Delta p_i} \right\rfloor \cdot \Delta e_i$. Nach (3.18) ist die in diesem Intervall mindestens zugeteilte Rechenzeit:

$$\left\lfloor \frac{t_d - t_1}{\Delta p_{sv}} \right\rfloor \cdot \Delta e_{sv} + \max(0, (t_d - t_1) \bmod \Delta p_{sv} - \Delta e_s - \Delta e_p)$$

Da es zu einer Fristverletzung zum Zeitpunkt t_d kommt, gilt:

$$\sum_{i=1}^{n} \left\lfloor \frac{t_d - t_1}{\Delta p_i} \right\rfloor \cdot \Delta e_i > \left\lfloor \frac{t_d - t_1}{\Delta p_{sv}} \right\rfloor \cdot \Delta e_{sv} + \max(0, (t_d - t_1) \bmod \Delta p_{sv} - \Delta e_s - \Delta e_p)$$

Dies steht im Widerspruch zu Ungleichung (3.19), von deren Gültigkeit ausgegangen wurde. Somit ist Ungleichung (3.19) auch ein hinreichendes Kriterium für die Planbarkeit der Prozesse.

Die rechte Seite der Ungleichung stellt die zugewiesene Rechenzeit zum Zeitpunkt t dar:

$$alloc(t) := \left\lfloor \frac{t}{\Delta p_{sv}} \right\rfloor \cdot \Delta e_{sv} + \max(0, t \bmod \Delta p_{sv} - \Delta e_s - \Delta e_p)$$

bzw. da $\Delta p_{sv} = \Delta e_{sv} + \Delta e_s + \Delta e_p$:

$$alloc(t) := \left\lfloor \frac{t}{\Delta p_{sv}} \right\rfloor \cdot \Delta e_{sv} + \max(0, t \bmod \Delta p_{sv} - \Delta p_{sv} + \Delta e_{sv}) \qquad (3.20)$$

Die linke Seite gibt den Rechenzeitbedarf der Prozessmenge P zum Zeitpunkt t an:

$$req(t) := \sum_{i=1}^{n} \left\lfloor \frac{t}{\Delta p_i} \right\rfloor \Delta e_i \qquad (3.21)$$

Planbarkeit ist gegeben, wenn $\forall t, t \geq 0$ gilt:

$$req(t) \leq alloc(t)$$

oder:

$$slack(t) := alloc(t) - req(t) \geq 0 \tag{3.22}$$

Die Funktion $slack(t)$ gibt die Differenz zwischen zugewiesener und benötigter Rechenzeit, also den bestehenden Spielraum an. Dieser muss für alle $t > 0$ größer oder gleich Null sein. Die Menge der Werte für t, für die Gültigkeit dieses Kriteriums zu zeigen ist, ist nach den bisherigen Überlegungen unbeschränkt, womit das Kriterium nicht praktikabel wäre. Daher soll diese Wertemenge im Folgenden eingeschränkt werden.

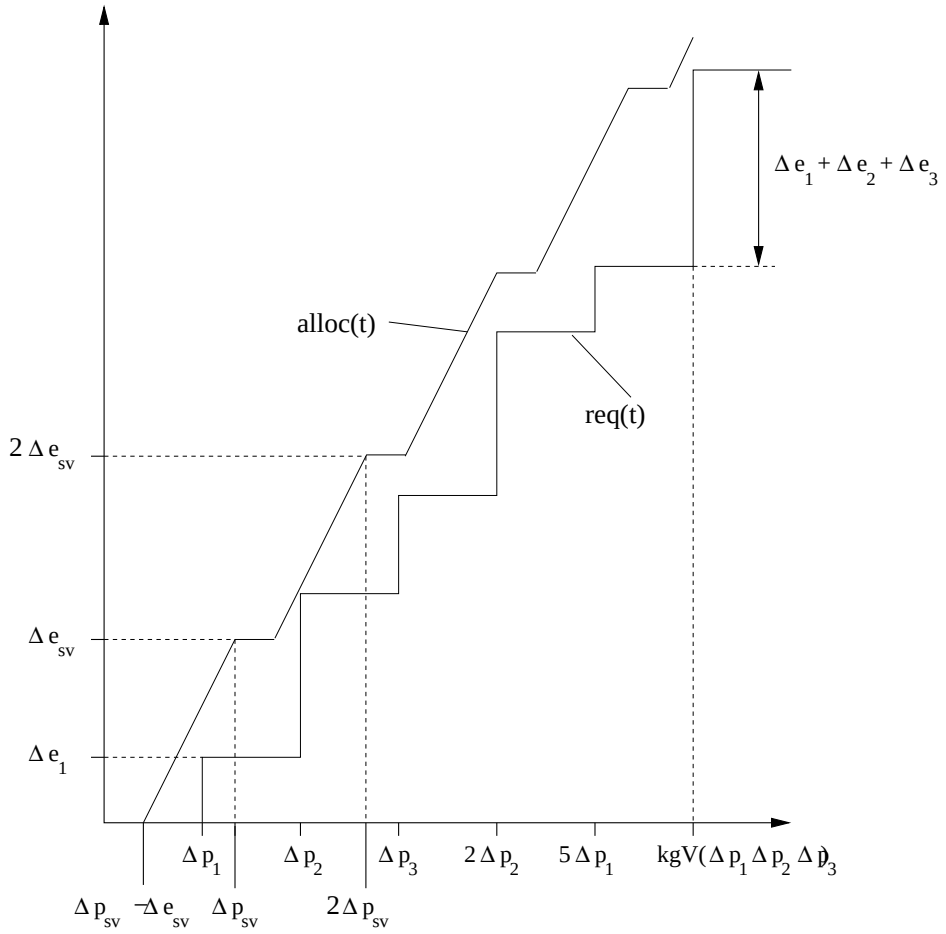

Abbildung 3.7: Zuweisung $alloc(t)$ und Bedarf $req(t)$ bei Planung nach EDF.

Abbildung 3.7 zeigt beispielhaft Verläufe der Funktionen $req(t)$ und $alloc(t)$. Beide Funktionen sind monoton steigend. Die Funktion $req(t)$ steigt sprunghaft

an, wobei alle Sprungstellen Vielfache der Periodendauern Δp_i sind. Nur an solchen Sprungstellen kann es vorkommen, dass der Bedarf $req(t)$ die zugewiesene Rechenzeit $alloc(t)$ übersteigt. Das Kriterium (3.19) muss somit nur für diese Sprungstellen geprüft werden[1].

Weiter ist festzustellen, dass die Funktion $alloc(t)$ periodisch ist, und zwar mit der Periodendauer Δp_{sv}:

$$alloc(t + \Delta p_{sv}) = alloc(t) + alloc(\Delta p_{sv}) \forall t \geq 0 \tag{3.23}$$

Die Funktion $req(t)$ ist ebenfalls periodisch, und zwar mit der Periodendauer Δp_{hyp}, dem kleinsten gemeinsamen Vielfachen der Periodendauern Δp_i aller Prozesse der Prozessmenge P (vgl. [ZÖ8], S. 178):

$$\Delta p_{hyp} = kgV(\{\Delta p_i | i \in P\}) \tag{3.24}$$

$$req(t + \Delta p_{hyp}) = req(t) + req(\Delta p_{hyp}) \forall t \geq 0 \tag{3.25}$$

Die Funktion $slack(t)$ ist als Überlagerung der Funktionen $alloc(t)$ und $req(t)$ somit ebenfalls periodisch. Ihre Periodendauer Δp_{slack} ist das kleinste gemeinsame Vielfache von Δp_{sv} und Δp_{hyp}. Damit ist der Wertebereich von t, für den die Gültigkeit des Kriteriums (3.19) zu zeigen ist, beschränkt auf $0 \leq t \leq \Delta p_{slack}$, wobei:

$$\Delta p_{slack} = kgV(\Delta p_{sv}, \Delta p_{hyp}) = kgV(\{\Delta p_i | i \in P\}, \Delta p_{sv}) \tag{3.26}$$

Die Anzahl der Werte für t, für die das Kriterium (3.19) für eine gegebene Prozessmenge zu prüfen ist, ist damit endlich, d.h. die Prüfung auf Planbarkeit ist prinzipiell durchführbar. Trotz der Beschränktheit kann die Menge zu prüfender Werte für t jedoch sehr groß und die Prüfung dementsprechend aufwändig werden. Im Folgenden werden daher einige weitere, leichter prüfbare Kriterien ermittelt.

So folgt aus (3.23) und (3.25), dass sowohl für $alloc(t)$ als auch für $req(t)$ eine mittlere Steigung angegeben werden kann. Da beide Funktionen für $t = 0$ den Wert 0 annehmen, ergibt sich die Steigung einfach aus dem Verhältnis des Wertes der jeweiligen Funktion am Ende ihrer ersten Periode und der Dauer dieser Periode. Die mittlere Steigung von $alloc(t)$ ist also:

[1] Dieser anhand von Abb. 3.7 offensichtliche Sachverhalt kann auch formal bewiesen werden. Für diesen formalen Beweis sei auf [Bol97] verwiesen.

$$U_{alloc} = \frac{alloc(\Delta p_{sv})}{\Delta p_{sv}} = \frac{\Delta e_{sv}}{\Delta p_{sv}}$$

Die mittlere Steigung von $req(t)$ ist:

$$U_{req} = \frac{req(\Delta p_{hyp})}{\Delta p_{hyp}} = \frac{\sum_{i=1}^{n} \left\lfloor \frac{\Delta p_{hyp}}{\Delta p_i} \right\rfloor \Delta e_i}{\Delta p_{hyp}}$$

Da $\Delta p_{hyp} = kgV(\{\Delta p_i | i \in P\})$ ist das Ergebnis der Division $\frac{\Delta p_{hyp}}{\Delta p_i}$ für alle $i \in P$ ganzzahlig, sodass die Abrundungs-Klammern ($\lfloor\rfloor$) weggelassen werden können. Damit ist:

$$U_{req} = \frac{\sum_{i=1}^{n} \frac{\Delta p_{hyp}}{\Delta p_i} \Delta e_i}{\Delta p_{hyp}} = \sum_{i=1}^{n} \frac{\Delta e_i}{\Delta p_i}$$

Dies ist zugleich die Auslastung durch die Prozessmenge P, $U(P)$ (vgl. 2.2.1).

Ist die mittlere Steigung der Zuweisung $alloc(t)$ kleiner als die des Bedarfs $req(t)$, so wird für $t \to \infty$ immer $alloc(t) < req(t)$, d.h. Kriterium (3.19) wird verletzt. Also ist ein weiteres Kriterium für die Planbarkeit:

$$U(P) = \sum_{i=1}^{n} \frac{\Delta e_i}{\Delta p_i} \leq \frac{\Delta e_{sv}}{\Delta p_{sv}} \tag{3.27}$$

Dieses Kriterium ist notwendig, aber nicht hinreichend. Es kann somit einem Planbarkeitstest nach Kriterium (3.19) „vorgeschaltet" werden, um nicht planbare Situationen schnell zu erkennen.

Abbildung 3.8 zeigt die gleichen Beispiele für Verläufe der Funktionen $req(t)$ und $alloc(t)$ wie Abbildung 3.7, jedoch in gröberem Maßstab. Daraus wird deutlich, dass, wenn die mittlere Steigung von $req(t)$ kleiner als die von $alloc(t)$ ist, die Verläufe der beiden Funktionen sich zunehmend voneinander entfernen. Ab einem bestimmten Punkt $t = t_{max}$ kann eine Verletzung des Planbarkeitskriteriums (3.19) mit Sicherheit ausgeschlossen werden, d.h. $\forall t, t > t_{max}$ ist das Kriterium immer erfüllt. Je nach der Größe der Unterschiedes zwischen den beiden mittleren Steigungen kann der Wert von t_{max} wesentlich kleiner sein, als die Obergrenze

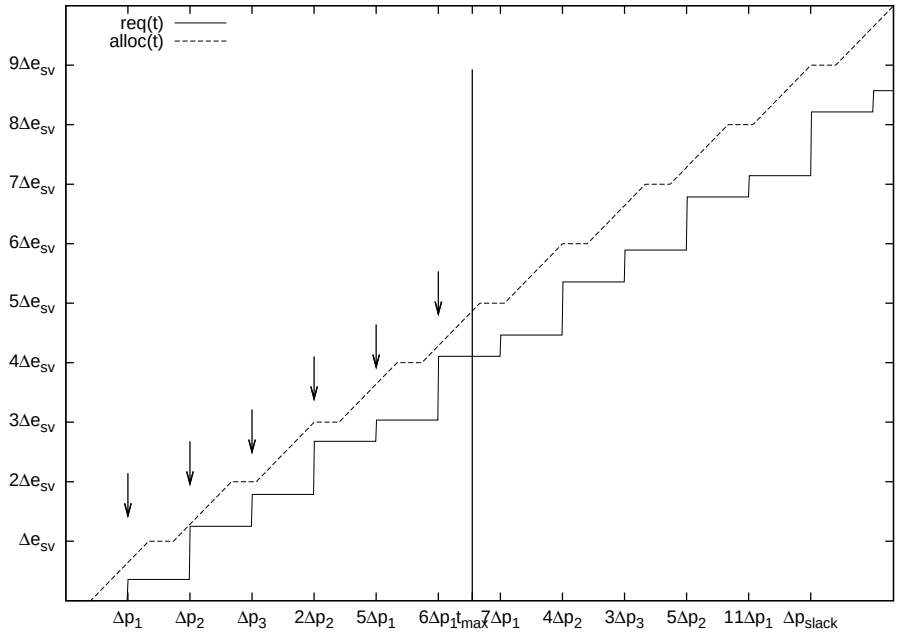

Abbildung 3.8: Zuweisung $alloc(t)$ und Bedarf $req(t)$ bei Planung nach EDF, grober Maßstab.

Δp_{slack} (vgl. Gleichung (3.26)), bis zu der nach den bisherigen Überlegungen die Gültigkeit des Kriteriums zu zeigen ist. Somit kann die Menge der zu prüfenden Werte von t u.U. erheblich reduziert werden. In [Bol97] wird allgemein gezeigt, dass gilt:

$$t_{max} = \frac{\Delta e_p + \Delta e_s}{1 - (U(P) + U_p + U_s)} = \frac{\Delta p_{sv} - \Delta e_{sv}}{1 - (U(P) + U_p + U_s)} \tag{3.28}$$

Darin ist $U(P)$ wieder die Auslastung durch die Prozessmenge P (vgl. 2.2.1), U_s und U_p sind die Auslastungen durch die Störprozesse δ_s und δ_p.

Für das in Abbildung 3.8 dargestellte Beispiel ist $\Delta p_{slack} = kgV(3, 6, 9, 4) = 36$ und $t_{max} = 19,636$, d.h. nur für die Vielfachen der Δp_i, die kleiner als t_{max} sind, ist das Planbarkeitskriterium (3.19) zu prüfen. Im Beispiel sind dies die mit Pfeilen bezeichneten sechs Werte, während ansonsten alle zwölf Vielfachen der Δp_i die kleiner oder gleich Δp_{slack} sind, geprüft werden müssten. Die Anzahl der zu prüfenden Werte wird bei diesem Beispiel also halbiert. Würde jedoch Δp_{sv} nur um 1% verringert, so wäre $\Delta p_{slack} = 2394$[1], während t_{max} mit $18,93$ nahezu

[1] Da ein kleinstes gemeinsames Vielfaches nur für ganze Zahlen angegeben werden kann, wurde hier mit Zeiteinheiten von 100 gerechnet, d.h. $\Delta p_{slack} = \frac{kgV(300,600,900,399)}{100} = 2394$

unverändert bliebe, d.h. nach der bisherigen Regel müssten nun 798 Werte geprüft werden, während es nach der neuen Regel nach wie vor genügt, die ersten sechs zu prüfen.

Mit dem Planbarkeitskriterium (3.19) und den vorgestellten Regeln zur Einschränkung der Menge von Werten, für die dieses Kriterium zu prüfen ist, steht nun ein brauchbares Instrumentarium zur Feststellung der Planbarkeit einer Prozessmenge nach EDF innerhalb eines Stellvertreterprozesses zur Verfügung. Für das Einplanen dieses Stellvertreterprozesses durch einen globalen Scheduler wird nun noch eine Beschreibung des Stellvertreterprozesses als Ganzes gesucht.

Wie gezeigt wurde, ist die Zuweisung, also die Menge an Rechenkapazität, die der Stellvertreterprozess benötigt, damit er seine Prozesse nach EDF planen kann, als Funktion $alloc(t)$ (Gleichung (3.20)) gegeben. Diese Funktion beschreibt einen periodischen Prozess mit der Periodendauer Δp_{sv} und der Ausführungszeit Δe_{sv}, d.h., wie beim zuvor beschriebenen Planen nach monotonen Raten kann auch ein Stellvertreterprozess, der intern nach Fristen plant, nach außen als periodischer Prozess beschrieben werden. Ein globaler Scheduler kann diesen Stellvertreterprozess seinerseits nach einem geeigneten Verfahren (z.B. EDF oder RMS) einplanen.

Nun fehlt noch ein Verfahren, nach dem ein Wertepaar $(\Delta e_{sv}, \Delta p_{sv})$, zur Beschreibung eines Stellvertreterprozesses aus den Parametern der Prozessmenge, die unter dem lokalen Scheduler des Stellvertreterprozesses arbeitet, ermittelt werden kann. Anders als beim Planen nach monotonen Raten gibt es hier keine harte Grenze des Wertebereiches von Δp_{sv}. Für eine gegebene Prozessmenge lassen sich beliebig viele Wertepaare $(\Delta e_{sv}, \Delta p_{sv})$ angeben, die Ungleichung (3.19) für alle anhand der vorgenannten Regeln zu prüfenden Werte von t erfüllen. Alle diese Wertepaare führen somit zu brauchbaren Plänen. Es wird ein weiteres Kriterium benötigt, anhand dessen aus der Vielzahl brauchbarer Pläne der „beste" auszuwählen ist.

Durch das Ausführen der Prozesse in einem periodischen Zeitfenster kann es –wie beim Planen nach monotonen Raten– zu Planungslücken kommen, was sich darin äußert, dass die Auslastung durch den Stellvertreterprozess größer sein kann, als die Summe der Auslastungen durch die gekapselten Prozesse (vgl. Gleichung (3.27)). Die „Güte" eines brauchbaren Planes kann daher an der Differenz zwischen diesen beiden Auslastungen gemessen werden. Für alle brauchbaren Wertepaare $(\Delta e_{sv}, \Delta p_{sv})$ wäre somit die Differenz:

$$\Delta U = U_{alloc} - U_{req} = \frac{\Delta e_{sv}}{\Delta p_{sv}} - \sum_{i=1}^{n} \frac{\Delta e_i}{\Delta p_i} \tag{3.29}$$

Prozessmenge	U_{req}	Δe_1	Δp_1	Δe_2	Δp_2	Δe_3	Δp_3	Δe_4	Δp_4
P_1	0.6389	1	3	1.5	6	0.5	9		
P_2	0.6389	0.1	3	0.5	6	4.7	9		
P_3	0.6889	0.1	3	0.5	6	4.7	9	1	20

Tabelle 3.1: Parameter der Prozessmengen in Abbildung 3.9.

zu bilden. Das Wertepaar, das zur geringsten Differenz ΔU führt, wäre zu wählen.

Aus Gleichung (3.20) ist ersichtlich, dass die Zuweisung $alloc(t)$ bei gleichbleibender Periodendauer Δp_{sv} und wachsender Ausführungszeit Δe_{sv} für beliebige t stets ansteigt. Der Bedarf $req(t)$ (siehe Gleichung (3.21)) ist hingegen unabhängig von Δe_{sv}. Gibt es also ein Paar $(\Delta e_{sv}, \Delta p_{sv})$, das Ungleichung (3.19) erfüllt und das somit einen brauchbaren Plan liefert, so liefert auch jedes weitere Paar $(\Delta e_{sv} + \epsilon, \Delta p_{sv})$ mit $0 \leq \epsilon < \Delta p_{sv} - \Delta e_{sv}$ einen brauchbaren Plan. Die „Güte" dieses Plans ist jedoch geringer, da nach Gleichung (3.29) die Auslastungsdifferenz für wachsende Δe_{sv} ansteigt. Somit ist für eine gewählte Periodendauer Δp_{sv} stets der kleinst-mögliche Wert für Δe_{sv} zu wählen, für den das Planbarkeitskriterium noch erfüllt ist, da so die Auslastungsdifferenz ΔU minimal wird.

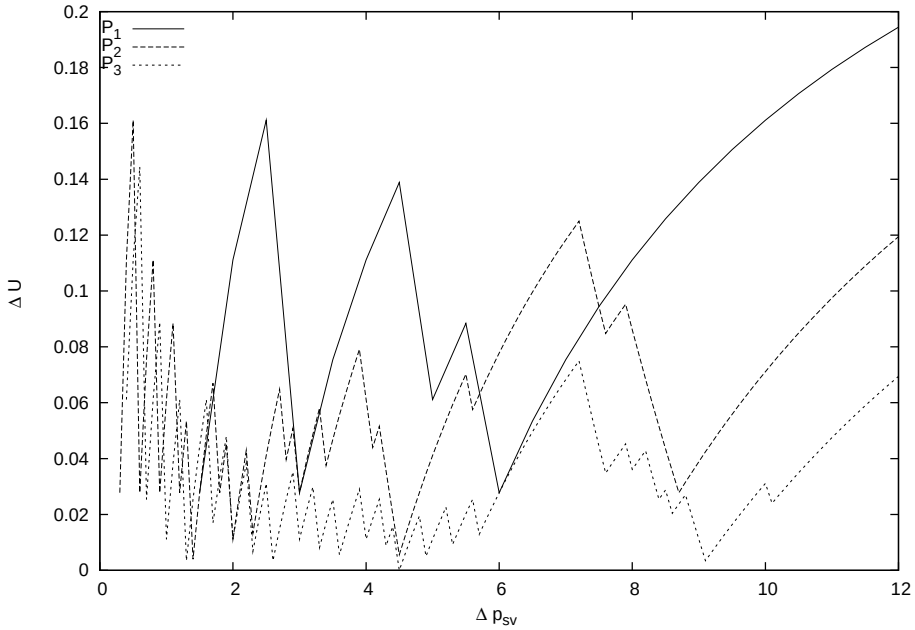

Abbildung 3.9: Auslastungsdifferenz ΔU als Funktion der Periodendauer.

Abbildung 3.9 zeigt einige von einem Hilfsprogramm (siehe Anhang B) errechnete Beispiele für Verläufe der Auslastungsdifferenz ΔU als Funktion der Perioden-

dauer Δp_{sv}. Dazu wurden für drei verschiedene Prozessmengen alle Periodendauern Δp_{sv} und die jeweils kleinstmöglichen Ausführungszeiten Δe_{sv} ermittelt, für die sich ein brauchbarer Plan ergibt. Die Parameter der betrachteten Prozessmengen sind in Tabelle 3.1 angegeben. Die Prozessmengen P_1 und P_2 besitzen gleiche Periodendauern und bewirken gleiche Auslastungen, doch unterscheiden sich die Ausführungszeiten der einzelnen Prozesse. Prozessmenge P_3 entsteht aus P_2 durch Hinzufügen eines weiteren Prozesses. In allen drei Fällen zeigt der Verlauf der Auslastungsdifferenz deutliche relative Minima, die somit geeignete Werte für Δp_{sv} liefern. Im Fall der Prozessmenge P_3 erreicht die Auslastungsdifferenz bei $\Delta p_{sv} = 4,5$ sogar den Wert 0, d.h. bei dieser Konstellation entstehen keine Planungslücken.

Somit liegt nun auch für das Planen nach Fristen ein Regelwerk vor, mit dessen Hilfe für eine Menge P periodischer Prozesse, die durch einem Stellvertreterprozess gekapselt sind, die Planungsdaten dieses Stellvertreterprozesses derart ermittelt werden können, dass er seinerseits als periodischer Prozess eingeplant werden kann. Seine Periodendauer kann über weite Bereiche frei gewählt werden, wobei sich jedoch unterschiedliche Güten des Plans ergeben.

Bei der Wahl einer geeigneten Periodendauer können auch praktische Kriterien eine Rolle spielen, die in der bisherigen Betrachtung nicht berücksichtigt wurden. So sollte der Wert nicht kleiner als nötig sein, da sonst die Umschaltfrequenz des Stellvertreterprozesses unnötig hoch wird, was erhöhte Umschaltverluste zur Folge hat. Abbildung 3.9 zeigt, dass einige der Minima der Auslastungsdifferenz unterhalb der kleinsten Periodendauer der Prozessmenge liegen, während sich andere, ähnlich niedrige Minima auch im Bereich zwischen der kleinsten und der größten Periodendauer finden. Um erhöhte Umschaltverluste zu vermeiden, kann es unter Umständen von Vorteil sein, eines dieser höheren Minima zu bevorzugen, auch wenn möglicherweise der errechnete Betrag der Auslastungsdifferenz dabei geringfügig höher liegt.

3.1.4 Kapselung von Nicht-Echtzeitprozessen

In Abschnitt 2.2.2 wurden Nicht-Echtzeitprozesse eingeführt, und es wurde festgestellt, dass diese nicht nach den Methoden der Echtzeitplanung behandelt werden können. Vielmehr werden für solche Prozesse andere, teilweise auf Heuristiken beruhende Planungsverfahren verwendet, die eine „faire" Verteilung der zur Verfügung stehenden Rechenleistung zum Ziel haben.

Verglichen mit der Echtzeitplanung können solche Verfahren zwar eine im Durchschnitt bessere Rechenleistung ermöglichen, aber sie können keine zeitlichen Garantien geben und sind daher nicht Gegenstand dieser Arbeit. Da es aber Ziel

dieser Arbeit ist, Echtzeit- und Nicht-Echtzeitprogramme gemeinsam in einem Rechensystem zu betreiben, muss zumindest die Möglichkeit bestehen, Nicht-Echtzeitprozesse bei der Echtzeitplanung mit zu berücksichtigen. Dazu definieren wir einen speziellen Prozesstyp N (Non-realtime), der, wie in 3.1 beschrieben, eine Menge von Nicht-Echtzeitprozessen und ihren lokalen Scheduler kapselt. Verschiedene Gruppen von Nicht-Echtzeitprozessen können dann als verschiedene Prozessobjekte N_i dieses Prozesstyps dargestellt werden. Wir nehmen an, dass jede derart gekapselte Menge von Nicht-Echtzeitprozessen immer mindestens einen gierigen Prozess enthält (als Minimum den erwähnten „idle"-Prozess), und dass der lokale Scheduler nach einer arbeitserhaltenden Methode vorgeht. Damit gibt es in jeder Ausführung N_i immer mindestens einen rechenbereiten Prozess, d.h. der Stellvertreterprozess ist seinerseits stets rechenbereit und somit ein nicht-endlicher, gieriger Prozess.

Nicht-Echtzeitprozesse sind in der Echtzeitplanung auf zwei Arten zu berücksichtigen:

1. Gegebenenfalls muss einer Gruppe von Nicht-Echtzeitprozessen eine vorgegebene, durchschnittliche Mindestmenge an Rechenkapazität zugewiesen werden, damit für sie ein bestimmter durchschnittlicher Mindestfortschritt gewährleistet werden kann.

2. Zusätzlich können Nicht-Echtzeitprozesse jederzeit als „Konsumenten" für in der Echtzeitplanung anfallende, überschüssige Rechenkapazitäten auftreten: Die Echtzeitplanung stellt sicher, dass alle Echtzeitprozesse über genügend Rechenzeit verfügen, um ihre jeweiligen Fristen einhalten zu können. In der Planung wird dabei von „worst-case"-Annahmen ausgegangen, die in der Mehrzahl der Fälle nicht eintreten. In der Folge verfügen Echtzeitprozesse über große Reserven an Rechenkapazität, die sie nur sehr selten vollständig ausschöpfen. Bei Systemen, in denen ausschließlich Echtzeitprozesse vorkommen, wird die daraus resultierende schlechte Prozessorausnutzung gemeinhin als „Preis der Echtzeitverarbeitung" hingenommen. Existieren im System jedoch auch Nicht-Echtzeitprozesse, so können diese die erst zur Laufzeit frei werdenden Rechenkapazitäten sinnvoll nutzen

Zur Beschreibung einer durchschnittlich verfügbaren Rechenkapazität für Nicht-Echtzeitprozesse ist die durchschnittliche Prozessorauslastung, die dieser Prozess verursacht, eine geeignete Größe. Ein solcher Prozess wird also gegenüber einem globalen Scheduler durch eine Prozessorauslastung U_{min} charakterisiert, die ihm mindestens gestattet werden muss. Da die hier betrachteten Nicht-Echtzeitprozesse gierig sind, würden sie bei ungehindertem Zugriff auf einen Prozessor

diesen stets zu 100 % auslasten. Es ist daher Aufgabe eines globalen Schedulers, diese Auslastung zu begrenzen, wobei die minimale Auslastung U_{min} nicht unterschritten werden darf. Der minimale Ausführungszeitbedarf eines Nicht-Echtzeitprozesses zum Zeitpunkt t ist damit gegeben durch:

$$req(t) = U_{min} \cdot t \tag{3.30}$$

Dies entspricht der Vorstellung der anteiligen Prozessorzuteilung, die in 2.2.8 beschrieben wurde. Wie dort bereits festgestellt wurde, ist die idealisierte kontinuierliche Verteilung praktisch nicht machbar, d.h. es muss eine endliche Periodendauer Δp so gewählt werden, dass dem Prozess während dieses Zeitintervalls mindestens die geforderte kumulierte Rechenzeit $\Delta e = U_{min} \cdot \Delta p$ zugeteilt wird.

Da Nicht-Echtzeitprozesse per Definition keine Fristen einzuhalten haben, könnte diese Periodendauer Δp im Prinzip beliebig groß gewählt werden, solange der Anteil der Ausführungszeit Δe entsprechend mitwächst. In der Praxis gibt es meist aber doch einige Fristen einzuhalten, die beispielsweise dadurch gegeben sein können, dass das System mit einem (nur begrenzt geduldigen) menschlichen Benutzer interagiert. Diese Fristen liegen aber um mehrere Größenordnungen über denen, die Echtzeitprozessen gewöhnlich abverlangt werden, und sie sind als eher „weiche" Anforderungen anzusehen, da auch die Überschreitung einer Frist tolerabel ist, so lange sie nicht zu häufig vorkommt. Die Obergrenze für akzeptable Werte der Periodendauer Δp ist damit durch diese „weichen" Fristen grob eingegrenzt. Eine (ebenfalls grobe) Eingrenzung nach unten ergibt sich aus den in Abschnitt 3.2 näher betrachteten Prozesswechselkosten.

Nach Festlegung einer geeigneten Periodendauer Δp_{sv} innerhalb dieser Grenzen kann somit ein durch die minimale Auslastung U_{min} charakterisierter Stellvertreterprozess, der eine Menge von Nicht-Echtzeitprozessen kapselt, als unterbrechbarer periodischer Prozess beschrieben werden, für dessen Ausführungszeit gilt:

$$\Delta e_{sv} = U_{min} \cdot \Delta p_{sv} \tag{3.31}$$

3.2 Kosten der Virtualisierung

Die Virtualisierung ist mit Laufzeitkosten verbunden und sie beeinflusst das Zeitverhalten der in den virtuellen Maschinen arbeitenden Prozesse. Insbesondere die Laufzeitkosten sind schwer zu formalisieren und nur in seltenen Ausnahmefällen berechenbar. In diesem Abschnitt wird daher ein Modell zur praktikablem *Abschätzung* dieser Laufzeitkosten vorgestellt und anhand praktischer Messungen evaluiert.

3.2.1 Rechenarbeit und Rechenleistung

Bei den in Abschnitt 2.2.1 eingeführten Verfahren zur Echtzeitplanung werden Arbeitsvolumina, d.h. Mengen an von einem Prozessor zu verrichtender oder verrichteter Arbeit in der Regel über Zeitdauern beschrieben. Dahinter steht die Annahme, dass ein Prozessor in gleichen Zeitintervallen stets die gleiche Menge an Arbeit verrichtet, dass seine *Rechenleistung* [1] also konstant ist. Bei den meisten heute eingesetzten Prozessoren trifft dies jedoch nicht zu: Durch die verwendeten Caches ändert sich die Leistung ständig in Abhängigkeit von der Lokalität der Daten und des gerade ausgeführten Programms. Darüber hinaus arbeiten heute viele Prozessoren mit variabler Taktfrequenz. Im Allgemeinen muss also davon ausgegangen werden, dass die Leistung eines Prozessors zeitlichen Schwankungen unterliegt.

Die Arbeit eines Prozessors besteht darin, Programme auszuführen, also durch Programmcode fortzuschreiten. Demnach entspricht seine Leistung der Geschwindigkeit, mit der dieser Fortschritt stattfindet. Wird diese zeitabhängige *Fortschrittsrate* mit $r(t)$ bezeichnet, so ist die Arbeit, die ein Prozessor während eines Zeitintervalls $[t_0, t_1]$ verrichtet:

$$W(t_0, t_1) = \int_{t_0}^{t_1} r(\tau) d\tau \tag{3.32}$$

Eine Ermittlung des zeitlichen Verlaufes der Fortschrittsrate ist in der Regel nicht praktikabel. Sie hängt von einer Vielzahl von Faktoren (z.B. ausgeführter Code, Zustand des ausgeführten Prozesses, Zustand des Prozessors, Taktfrequenz, Speicherzugriffsgeschwindigkeit, etc.) ab, die in den meisten Fällen nicht oder nicht in ausreichendem Detail bekannt sind. Selbst wenn all diese Informationen gegeben wären, so wäre eine analytische Ermittlung –etwa mit Hilfe eines vollständigen Modells des Rechensystems– deutlich aufwändiger, als die eigentliche Ausführung des Codes selbst und somit im Zuge einer dynamischen Echtzeitplanung nicht wirtschaftlich. Bei statischer Echtzeitplanung, d.h. bei einer „off-line" Ermittlung der Fortschrittsrate kann die Berechnung dann möglich und wirtschaftlich sein, wenn sich die Menge zu berücksichtigender Eingangszustände auf eine hinreichend niedrige Anzahl reduzieren lässt. Auch dies ist nur in Ausnahmefällen möglich.

Diese Größe wird für eine Abschätzung der Verluste, wie sie in einem Rechensystem durch Prozessunterbrechungen entstehen, benötigt. Für diese Überlegungen

[1] $Leistung = \frac{Arbeit}{Zeit}$

genügt die Vorstellung einer *mittleren Forschrittsrate*, die ggf. auch empirisch ermittelt werden kann. Die mittlere Forschrittsrate ist diejenige konstante Rate, bei der innerhalb eines Intervalls $[t_0, t_1]$ die gleiche Menge an Arbeit verrichtet würde wie mit der tatsächlichen, variablen Fortschrittsrate:

$$\bar{r}(t_0, t_1) = \frac{W(t_0, t_1)}{t_1 - t_0} = \frac{\int_{t_0}^{t_1} r(\tau) d\tau}{t_1 - t_0} \tag{3.33}$$

Damit ist die von der Ausführung eines nicht unterbrechbaren Prozessobjekts P_i mit der Startzeit s_i und der Abschlusszeit c_i verrichtete Arbeit gleich:

$$W(s_i, c_i) = (c_i - s_i) \cdot \bar{r}(s_i, c_i) \tag{3.34}$$

Mit $\Delta e_i = c_i - s_i$ (nicht unterbrechbarer Prozess) und $\bar{r}_i := \bar{r}(s_i, c_i)$ (mittlere Fortschrittsrate des Prozessobjekts P_i) wird daraus:

$$W_i = \Delta e_i \cdot \bar{r}_i \tag{3.35}$$

Unter Vernachlässigung der durch Unterbrechungen bedingten Verluste gilt dies auch für unterbrechbare Prozesse, deren gesamte Ausführungszeit Δe_i beträgt (Siehe Abbildung 2.5).

3.2.2 Kosten des Kontextwechsels

Generell wird auf einem realen Rechensystem die mittlere Fortschrittsrate eines Prozesses am höchsten sein, wenn er ohne Unterbrechungen ausgeführt wird. Kommt es hingegen zu Unterbrechungen, so gehen diese mit zusätzlichen Kosten einher. Eine analytische Bestimmung dieser Verluste scheitert in der Regel an den bereits im vorangegangenen Abschnitt erwähnten Problemen bei der Bestimmung der Fortschrittsrate. Deshalb werden diese Kosten bei theoretischen Überlegungen zur Prozessplanung häufig ignoriert. Dabei unterscheiden sich aber verschiedene Planungsverfahren gerade in der Anzahl an Umschaltvorgängen, die sie –bei gleicher Problemlage– vornehmen. Um diese Verfahren besser vergleichen zu können, ist zumindest eine grobe Einschätzung der bei der Umschaltung wirkenden Mechanismen und ihrer Auswirkungen erforderlich. Darüber hinaus spielen in dieser Arbeit die Kosten von Unterbrechungen und Umschaltungen eine wichtige Rolle, da sie die Grenze für die in virtuellen Maschinen erzielbaren Echtzeitleistungen entscheidend beeinflussen. Im Folgenden wird daher eine Methode zur Abschätzung dieser Kosten entwickelt. Damit ist zwar keine exakte Voraussage für einen

konkreten Fall möglich, jedoch genügt diese Einschätzung in vielen Fällen, um z.B. in einer Echtzeitplanung die Kosten in Form von Aufschlägen auf die zugewiesenen Rechenzeitkontingente berücksichtigen zu können.

Bei der Ausführung mehrerer unterbrechbarer Prozesse auf einem Rechensystem sind zwei Arten von Verlusten zu unterscheiden:

1. Unterbrechungsverluste: Hiermit sind solche Aufwände gemeint, die durch die Unterbrechung eines in Ausführung befindlichen Prozesses anfallen. In der Regel werden solche Unterbrechungen über von einer Hardware (z.B. einer Uhr) erzeugte Interrupts ausgelöst. Die Bearbeitung eines solchen Interrupts führt unmittelbar zu Kosten für das Retten und das erneute Laden von Prozessorregistern, den Übergang in den privilegierten Modus, die spätere Rückkehr in den nicht-privilegierten Modus, das Wiederherstellen des Prozessorzustandes, den Wechsel des Adressraums und –nicht zuletzt– das Ausführen des Schedulers, der u.U. eine Prozessumschaltung vornimmt. Man beachte, dass dieser Aufwand unabhängig von den Entscheidungen des Schedulers ist: ob der Zustand des gerade unterbrochenen Prozesses oder der eines anderen, früher unterbrochenen Prozesses wiederhergestellt wird, der Aufwand für das Laden der Prozessorregister ist stets der gleiche.

2. Umschaltverluste: Diese Verluste entstehen indirekt als Folge des Wechsels in einen anderen Prozess. So führt zum Beispiel bei Systemen mit virtueller Adressierung eine im Zuge eines Wechsels vorgenommene Umschaltung des Adressraums dazu, dass der TLB-Cache des Prozessors mit neuen Inhalten geladen werden muss: Immer, wenn der neue Prozess Speicherinhalte referenziert, muss der Prozessor die zugehörige Abbildung der virtuellen Adresse auf die physische Adresse aus der Seitentabelle ermitteln, wobei er zusätzliche Buszyklen durchführt. Gleiches gilt für die stets vorhandenen Caches: nach dem Wechsel findet der neue Prozess die Caches infolge der Aktivitäten anderer Prozesse mehr oder weniger stark „abgekühlt" vor, d.h. sie enthalten zahlreiche von anderen Prozessen vorgenommene Einträge, die im Kontext des neuen Prozesses keine Gültigkeit haben. Mitunter ist es aus technischen Gründen oder aus Gründen der Datensicherheit sogar erforderlich, die Caches während eines Kontextwechsels explizit zu löschen.

Die Unterbrechungsverluste entstehen mit jeder Unterbrechung des laufenden Prozesses, unabhängig davon, ob die Unterbrechung zu einem Wechsel des laufenden Prozesses führt. Umschaltverluste entstehen hingegen nur im Zusammenhang mit Prozessumschaltungen. Eine Prozessumschaltung wird durch einen Aufruf des Schedulers eingeleitet, d.h. einer Prozessumschaltung geht immer eine Unterbrechung voraus. Umgekehrt führt aber nicht jede Unterbrechung zwangsläufig

zu einem Prozesswechsel: Zum Beispiel wird bei den in 2.2.8 beschriebenen Verfahren zur anteiligen Zuteilung der Scheduler regelmäßig in festen Zeitabständen aufgerufen, wobei mit jedem Aufruf neu entschieden wird, ob ein Prozesswechsel vollzogen werden soll. Daher müssen diese beiden Arten von Verlustbeiträgen getrennt betrachtet werden.

Unterbrechungsverluste

Die Unterbrechungsverluste manifestieren sich aus Sicht der Prozesse in Form eines Zeitfensters, während dessen keiner von ihnen fortschreitet. Die Dauer dieser Unterbrechung (engl. *break*), Δe_{br}, kann als annähernd konstant angesehen werden: Sie hängt von einer Reihe von Parametern ab, die sich während der Systemlaufzeit nicht ändern.

Wenn ein Prozess über ein Zeitintervall der Länge Δe_i ununterbrochen ausgeführt wird, so leistet er dabei die Arbeit $W_i = \Delta e_i \cdot \bar{r_i}$ (vgl. Gleichung (3.35)). Wird die Ausführung des Prozesses für die Dauer Δe_{br} unterbrochen, so ist demnach der damit einhergehende Verlust an Rechenarbeit:

$$W_{br} = \Delta e_{br} \cdot \bar{r_i} \tag{3.36}$$

In einer Echtzeitplanung können Unterbrechungsverluste auf zwei Arten berücksichtigt werden:

1. *Als globale Störprozesse*: Die Unterbrechungen werden als Ausführungen eines oder mehrerer imaginärer Prozesse aufgefasst. Bei regelmäßigen Unterbrechungen ist ein solcher Störprozess als periodisch, bei unregelmäßigen Unterbrechungen als aperiodisch bzw. sporadisch anzusehen. Diese Störprozesse werden zur betrachteten Prozessmenge hinzugefügt, um Unterbrechungsverluste zu modellieren.

2. *Als Zuschlag zur Ausführungszeit*: Die Ausführungszeiten der betrachteten Prozesse werden pro Unterbrechung um Δe_{br} erhöht.

Die erste Methode ermöglicht eine Einplanung der Verluste in einer Prozessplanung, sodass Aussagen über die Planbarkeit eines Prozesssystems unter Berücksichtigung der Umschaltverluste weiterhin möglich sind. Die zweite Methode erlaubt es, Verluste einzelnen Prozessen zuzuordnen, sodass, je nach den Anforderungen des jeweiligen Prozesses, individuell angepasste, optimistische oder pessimistische Methoden zur Schätzung dieser Verluste angewendet werden können.

Umschaltverluste

Im Gegensatz zu den Unterbrechungsverlusten äußern sich die Umschaltverluste in einer kontinuierlichen „Verlangsamung" des Programmfortschritts des jeweils arbeitenden Prozesses, da der Prozessor neben der eigentlichen Programmausführung zusätzliche Zyklen auf das Befüllen der Caches und des TLB verwenden muss. Diese Verluste entstehen also erst während der Ausführung von Prozessen. Ihre Höhe hängt von der Häufigkeit ab, mit der der Prozessor nicht im Cache[1] befindliche Daten über den externen Bus nachladen muss, also von der Häufigkeit der Cache-Fehler.

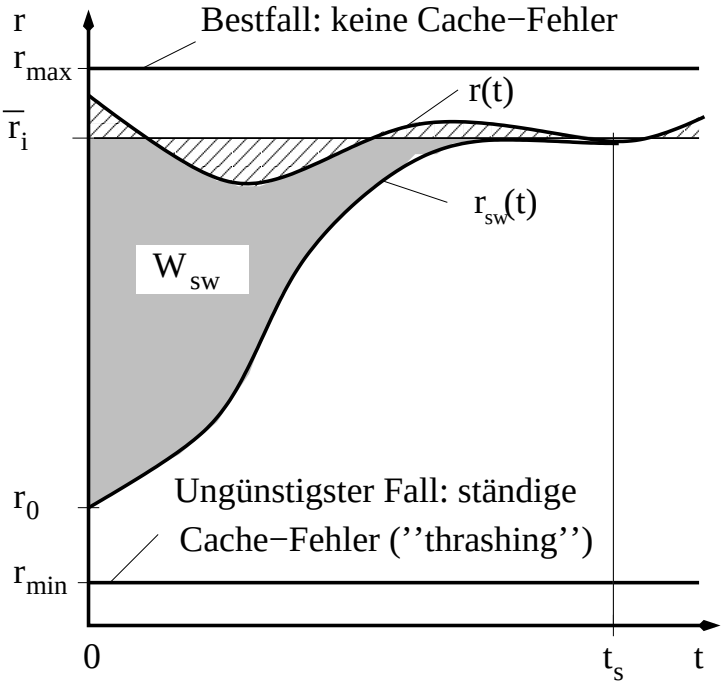

Abbildung 3.10: Verlauf der Fortschrittsrate nach einer Prozessumschaltung.

Abbildung 3.10 zeigt zwei mögliche Verläufe der Fortschrittsrate eines Prozesses. Dabei ist $r(t)$ der Verlauf bei kontinuierlicher, ununterbrochener Prozessausführung, und $r_{sw}(t)$ der Verlauf im Falle einer Prozessumschaltung zum Zeitpunkt $t = 0$. Zu diesem Zeitpunkt sind die Caches mehr oder weniger stark „abgekühlt", d.h. sie enthalten nur wenige oder keine Einträge, die für den nun aktiven Prozess von Nutzen wären. Infolgedessen ist die Ausführungsrate zunächst niedrig (r_0). Der bezüglich der Cache-Ausnutzung ungünstigste Fall ist das sogenannte „cache

[1] Im folgenden wird „Cache" als Synonym für die Gesamtheit aller Cache-Speicher eines Prozessors, einschließlich des TLB, verwendet.

thrashing": in diesem Fall werden Speicherinhalte vom Programm in derart unvorteilhafter Reihenfolge angefordert, dass jeder Cache-Eintrag verdrängt wird, bevor er ein zweites Mal genutzt werden könnte. Selbst in diesem Extremfall ist aber die Fortschrittsrate größer als 0, da ein Laden von Cache-Einträgen immer aufgrund von ausgeführten Befehlen geschieht, d.h. es gibt keine Cache-Fehler ohne Programmfortschritt. Diese minimale Fortschrittsrate (r_{min} in Abbildung 3.10) ist somit nicht zu unterschreiten, d.h. sie stellt zugleich die Untergrenze für mögliche Werte von r_0 dar.

Im Zuge des Programmfortschritts für $t > 0$ werden mehr und mehr Einträge in den Cache geladen, sodass sich der Verlauf von $r_{sw}(t)$ dem Verlauf von $r(t)$ annähert. Ab dem Zeitpunkt $t = t_s$ sind die beiden Verläufe gleich. Die durch den Umschaltvorgang bei $t = 0$ verlorene Rechenarbeit entspricht der Fläche zwischen den beiden Verläufen:

$$W_{sw}(t) = \int_0^t r(\tau)d\tau - \int_0^t r_{sw}(\tau)d\tau = t \cdot \bar{r_i} - \int_0^t r_{sw}(\tau)d\tau \tag{3.37}$$

Um wieder die Umschaltverluste durch eine äquivalente Zeitspanne auszudrücken, beziehen wir die Verlustarbeit auf die gesamte im Zeitintervall $[0, t]$ geleistete Arbeit ($t \cdot \bar{r_i}$) und multiplizieren mit der Länge des Intervalls (t):

$$\Delta e_{sw}(t) = t - \int_0^t \frac{r_{sw}(\tau)}{\bar{r_i}}d\tau \tag{3.38}$$

Während der Zeitspanne $0 \leq t \leq t_s$ wird der Prozessor gewissermaßen anteilig zwischen einer Nutzlast (dem ausgeführten Prozess) und einem imaginären Prozess, der das Laden der Caches übernimmt, genutzt. Wir definieren den *Nutzlastfaktor* als Anteil der Nutzlast an der insgesamt aufgebrachten Rechenleistung:

$$f(t) := \frac{r_{sw}(t)}{\bar{r_i}} \tag{3.39}$$

Damit wird aus Gleichung (3.38):

$$\Delta e_{sw}(t) = t - \int_0^t f(\tau)d\tau \tag{3.40}$$

Der konkrete Verlauf der Funktion $f(t)$ kann, wie bereits erwähnt, in der Regel nicht analytisch bestimmt werden. Allerdings ist es möglich, günstigste bzw.

ungünstigste Fälle zu identifizieren und zu beschreiben. Auch für den Bereich zwischen diesen beiden Extremen können Funktionen angegeben werden, die das Verhalten einer realen Situation approximieren. Der günstigste Fall ist zweifellos der, in dem keine Umschaltverluste entstehen, entweder (a) weil die Cache-Inhalte durch die Aktivitäten anderer Prozesse während der inaktiven Zeitspanne des betrachteten Prozesses nicht verändert wurden, d.h. der Prozess findet die Caches „so vor, wie er sie hinterlassen hat", oder (b) weil der Prozess keine oder nur sehr wenige Daten im Cache hält, sodass er durch den Cache keine nennenswerte Beschleunigung erfährt. In beiden Fällen wäre $f(t) = 1 \; \forall \; t$ und somit $\Delta e_{sw}(t) = 0$. Für den ungünstigsten Fall müsste die Fläche in Abbildung 3.10 maximal werden. Dazu müsste $f(t)$ konstant minimal sein und zu einem Zeitpunkt $t = t_s$ sprunghaft maximal werden. Dabei ist der Minimalwert gegeben durch den Fall des „Cache Thrashing", d.h. er ist $f_0 = \frac{r_{min}}{\bar{r_i}}$, und der Maximalwert ist wieder 1. Dieser Fall entspricht einer Sprungfunktion:

$$f_{flood}(t) = \begin{cases} f_0, & 0 \leq t < t_s \\ 1, & t \geq t_s \end{cases} \tag{3.41}$$

Ein Szenario, das ein solches Verhalten reproduzieren würde, wäre ein Prozess, der die Caches maximal „abgekühlt" vorfindet und sequenziell eine Menge von Datenobjekten referenziert, die in ihrer Gesamtheit vollständig in den Cache passen. Ein solcher „Cache-Fluter" würde so lange mit minimaler Rate fortschreiten, bis sich alle Datenobjekte im Cache befinden. Ab diesem Zeitpunkt würde er mit maximaler Rate fortschreiten. Diese Situation lässt sich programmtechnisch reproduzieren und messen, wodurch realistische Werte für die Parameter f_0 und t_s ermittelt werden können.

Gleichung (3.41) in Gleichung (3.40) eingesetzt ergibt:

$$\Delta e_{sw}^{flood}(t) = \begin{cases} t \cdot (1 - f_0), & 0 \leq t < t_s \\ t_s \cdot (1 - f_0), & t \geq t_s \end{cases} \tag{3.42}$$

Die Umschaltverluste hängen demnach von zwei Parametern ab:

1. t_s: Die Zeit, die der Prozess benötigt, um die Gesamtheit seiner Datenobjekte in den Cache zu laden. Es ist zu erwarten, dass diese Zeit proportional zur Größe dieser Datenmenge ist.

2. f_0: Das Verhältnis zwischen niedrigster und durchschnittlicher Fortschrittsrate. Dieser Wert gibt an, in welchem Umfang der betrachtete Prozess von den Caches profitiert. Er sollte abhängig vom ausgeführten Code und insbesondere unabhängig von der Datenmenge sein.

Der „Cache-Fluter" ist der ungünstigste Fall. Ausgehend von ihm werden sich stets sehr pessimistische Abschätzungen ergeben. Realistische Fälle müssen zwischen diesem und dem günstigsten Fall (d.h. keine Umschaltverluste) liegen.

Im Folgenden wird versucht, unter der Annahme eines exponentiellen Verlaufes des Nutzlastfaktors Aussagen über die zu erwartenden Verluste zu machen. In der Natur gibt es zahlreiche Beispiele für zufällige Vorgänge (z.B. den radioaktiven Zerfall), die nach einer Exponentialfunktion gegen einen Grenzwert konvergieren. Daher ist es naheliegend auch hier eine Exponentialfunktion als Abschätzung anzunehmen. Inwieweit diese Annahme tatsächlich ein geeignetes Modell zur Approximation des Verhaltens realer Systeme unter realen Lastbedingungen liefert, bleibt noch zu prüfen In jedem Fall ist aber davon auszugehen, dass die so gewonnenen Aussagen, selbst wenn sie nicht exakt sind, immer noch näher an der Realität liegen werden, als Aussagen, die unter der allzu oft praktizierten, gänzlichen Vernachlässigung von Umschaltverlusten gewonnen wurden.

Wir nehmen also an, der Nutzlastfaktor verlaufe unter gegebenen Lastbedingungen entsprechend einer Exponentialfunktion, die zwischen dem ungünstigstmöglichen Verlauf (d.h. dem des „Cache-Fluters") und dem günstigst-möglichen Verlauf liegt:

$$f_{avg}(t) = 1 + (f_0 - 1) \cdot e^{-kt} \tag{3.43}$$

wobei:

$$k = \frac{1}{t_s} \cdot ln\left(\frac{1 - f_0}{\epsilon}\right)$$

Diese Funktion konvergiert gegen 1 (d.h. $\lim_{t\to\infty} f_{avg}(t) = 1$), erreicht diesen Wert aber nicht in endlicher Zeit. Daher ist hier t_s definiert als die Zeit, zu der die Abweichung zum Endwert 1 weniger als die Differenz ϵ beträgt, d.h. $f_{avg}(t_s) = 1 - \epsilon$. Diese Funktion in Gleichung (3.40) eingesetzt ergibt:

$$\Delta e_{sw}^{avg}(t) = \frac{1 - f_0}{k \cdot t} \cdot (1 - e^{-kt}) \tag{3.44}$$

Auch hier ergibt sich eine Abhängigkeit von den beiden Parametern t_s und f_0 sowie von der Abweichung ϵ, die zur Definition von t_s benötigt wurde.

Die Gleichungen (3.42) und (3.44) erlauben bei Kenntnis dieser Parameter die Ermittlung der Kosten einer Prozessumschaltung in Form eines äquivalenten

Zeitaufschlages. Zur Berücksichtigung der Umschaltverluste können, wie bereits im vorigen Abschnitt bei den Unterbrechungsverlusten, entweder globale Störprozesse angenommen werden, oder die Ausführungszeiten der Nutzprozesse können für jeden Umschaltvorgang um den entsprechenden Aufschlag $\Delta e_{sw}(t)$ erhöht werden. Dabei ist für t die jeweilige Laufzeit des Prozesses bis zur nächsten Unterbrechung anzusetzen. Je länger diese (im Verhältnis zu t_s) ist, um so geringer wird der Anteil der Verluste.

Bei der Berücksichtigung der Verluste in Form eines Zeitaufschlages ergibt sich die Möglichkeit, die Verlustbeiträge jeweils einzelnen Prozessen zuzuordnen. Auf diese Weise können auf jeden betrachteten Prozess spezifische Abschätzungsfunktionen angewendet werden, was die Anpassung der Abschätzung an die individuellen Eigenschaften der Prozesse ermöglicht. So kann beispielsweise für Prozesse mit „harten" Zeitanforderungen (vgl.: 2.2) mit der (pessimistischen) Funktion $\Delta e_{sw}^{flood}(t)$ nach Gleichung (3.42) gearbeitet werden. Die so erhaltenen Schätzungen sind worst-case Werte, d.h. sie werden in der Regel unter-, nicht aber überschritten, sodass der Prozess seine Fristen sicher einhalten kann. Auf Prozesse mit „weichen" Zeitanforderungen kann hingegen die optimistischere Funktion $\Delta e_{sw}^{avg}(t)$ nach Gleichung (3.44) angewendet werden. Die so erhaltenen Schätzungen sind deutlich günstiger als die worst-case Werte, aber es besteht das Risiko, dass die Verluste von Fall zu Fall höher als vorhergesagt ausfallen, was dann zu (für „weiche" Echtzeit tolerierbaren) gelegentlichen Fristüberschreitungen führt.

3.2.3 Messungen

Eine Reihe von Messungen wurde durchgeführt, um das durch Gleichung (3.41) beschriebene Verhalten eines „Cache-Fluters" zu reproduzieren. Diese Messungen hatten zwei Ziele:

1. Zeigen, dass das vorgestellte Modell grundsätzlich anwendbar ist.
2. Ermitteln realistischer Zahlenwerte für die Parameter f_0 und t_s des Modells.

Messmethodik

Es wurde eine Methode benötigt, um den zeitlichen Verlauf der Fortschrittsrate eines Prozesses zu ermitteln. Ein naheliegender Weg zur Feststellung von „Programmfortschritt" wäre die Ermittlung der Rate, mit der ein Prozessor Maschinenbefehle ausführt. Allerdings sind verschiedene Maschinenbefehle sowohl hinsichtlich der Menge an „nützlicher Arbeit ", die sie verrichten, als auch hinsichtlich der Zeit, die zu ihrer Ausführung benötigt wird, nicht immer miteinander

vergleichbar. Es gibt schon innerhalb einer einzigen Prozessorarchitektur keinen allgemein gültigen „durchschnittlichen Maschinenbefehl", geschweige denn eine Maschinenoperation, die Vergleiche zwischen verschiedenen Architekturen erlauben würde.

Zur Schätzung der Umschaltverluste ist zudem die absolute Fortschrittsrate nicht erforderlich: In Gleichung (3.42) geht lediglich die *Nutzlastfunktion*, d.h. das Verhältnis aus tatsächlicher Fortschrittsrate und der durchschnittlichen Fortschrittsrate, wie sie ohne Umschaltverluste vorgelegen hätte, ein.

Bei der verwendeten Messmethode werden anstelle einzelner Maschineninstruktionen kurze Instruktionssequenzen verwendet, die bestimmte, einheitliche Mengen an Arbeit verrichten, wie zum Beispiel das Laden, Speichern oder Kopieren einer bestimmten Anzahl von Datenobjekten. Diese Instruktionssequenzen werden innerhalb enger Schleifen mit wählbarer Durchlaufzahl ausgeführt, sodass die kleinste messbare Arbeitsmenge in einem einzelnen Schleifendurchlauf besteht. Aufgrund der begrenzten Auflösung der zur Verfügung stehenden Bausteine zur Zeiterfassung, und da der programmtechnische Aufwand zum Auslesen dieser Bausteine gegenüber den zu messenden Instruktionssequenzen nicht vernachlässigbar ist, wird bei der Messung eine programmierbare Anzahl von Schleifendurchläufen ausgeführt, und die dafür benötigte Zeit wird ermittelt. Das Resultat einer solchen Messung entspricht mithin nicht einer Fortschritts*rate*, sondern es wird der innerhalb des gemessenen Zeitintervalls insgesamt gemachte Programmforschritt, d.h. die verrichtete Rechen*arbeit* ermittelt. Diese ist das Integral der Fortschrittsrate über die Zeit, sodass die Fortschrittsrate durch eine Ableitung nach der Zeit ermittelt werden kann.

Die eingesetzte Messmethode ist eine vereinfachte[1] Form der von John und Baumgartl in [JB06] beschriebenen Vorgehensweise: Unmittelbar vor der Durchführung einer Messung werden die Caches des Prozessors in einen definierten Zustand gebracht: Ihre Inhalte können wahlweise für ungültig erklärt werden („invalidate"), oder die Caches können mit Daten gefüllt werden, die von den nachfolgenden Testsequenzen nicht mehr referenziert werden. Dieses Befüllen der Caches geschieht durch Schreibzugriffe, sodass die Caches danach aus Sicht der Testsequenzen maximal „verunreinigt" sind, d.h. der Inhalt einer jeden Cache-Zeile muss vor ihrer Nutzung erst in den Speicher zurückgeschrieben werden.

Nachdem die Caches in dieser Weise vorbereitet wurden, wird eine ausgewählte Testsequenz für eine wählbare Anzahl von Schleifendurchläufen ausgeführt. Die

[1] John und Baumgartl verwenden noch eine weitere, von ihnen als „cache flooding" bezeichnete Methode, die hier nicht implementiert wurde: Mit „flooding" kann eine maximal ungünstige (jedoch nicht sehr realistische) Situation geschaffen werden, die nur in Verbindung mit der zweistufigen Cache-Struktur der IA-32 Architektur möglich ist.

dabei benötigte Rechenzeit wird als Ergebnis der Messung zurückgeliefert. Dieser Ablauf wird für unterschiedliche Anzahlen von Schleifendurchläufen wiederholt. Die Testsequenzen führen wahlweise Schreib- oder Lesezugriffe auf einen Datenbereich aus, der zuvor nicht in den Cache übernommen wurde. Die Größe dieses Datenbereiches ist wählbar. Sie entspricht der Menge an Datenobjekten, die ein betrachtetes Programm maximal in den Cache lädt. Sie wird im Folgenden als der „Arbeitsspeicher" des Programms bezeichnet[1]. Die Folge der Adressen innerhalb des Arbeitsspeichers, die von den Testsequenzen referenziert werden, kann wahlweise zufällig, oder zu linear, jeweils um die Größe einer Cache-Zeile aufsteigenden Adressen gewählt werden. Letzteres entspricht dem nachzubildenden „worst-case" Verhalten, d.h., es werden mit maximaler Rate Cache-Fehler so lange generiert, bis der gesamte Arbeitsspeicher in den Cache geladen wurde.

Als Testplattformen wurden zwei verschiedene IA-32-Maschinen sowie ein auf dem PowerPC MPC5200 basierendes Entwicklungs-Board unter Linux verwendet. Um das Ergebnis verfälschende Unterbrechungen der Test-Befehlssequenzen durch Interrupts zu vermeiden, wurden diese im privilegierten Modus mit maskierten Interrupts ausgeführt, weshalb es nötig war, den Test-Code in Form eines Kernmoduls in den Linux-Kern einzubinden.

Abbildung 3.11 zeigt beispielhaft die Ergebnisse von fünf Messungen, die mit unterschiedlichen Arbeitsspeichergrößen durchgeführt wurden. Im ersten (mit „Cache unverändert" bezeichneten) Fall wurde der Cache vor dem Test weder gefüllt noch gelöscht. Da jede Messung mehrfach wiederholt wurde, ist in diesem Fall davon auszugehen, dass alle im Anschluss durch die Testsequenz referenzierten Daten des Arbeitsspeichers zuvor bereits im Cache waren, sodass es hier zu keinen Cache-Fehlern kommt. In den übrigen vier Fällen wurde der Cache vor jeder Messung durch Schreibzugriffe gefüllt. Anschließend wurden Schreibzugriffe an aufsteigenden Adressen bis zur Größe des jeweils angegebenen Arbeitsspeichers durchgeführt. Die Kurven, die dabei für Arbeitsspeichergrößen von 16k, 32k und 64k aufgenommen wurden, folgen alle dem gleichen Muster: Die Anzahl der ausgeführten Schleifendurchläufe steigt zunächst langsam, bis zum Erreichen des Punktes, an dem der gesamte Arbeitsspeicher in den Cache geladen wurde. An diesem Punkt steigt die Fortschrittsrate schlagartig an. Wann dieser Übergang eintritt, hängt von der Größe des Arbeitsspeichers ab. Im Falle des 128k großen Arbeitsspeicher liegt der Übergang außerhalb des betrachteten Messintervalls. Im Falle „Cache unverändert" ist die Fortschrittsrate von Anfang an so hoch wie in den übrigen Fällen erst nach dem Übergang.

Die Kurven in Abbildung 3.12 wurden aus denen in Abbildung 3.11 durch Ab-

[1] Der gelegentlich in der Literatur zu findende Begriff „working set" wird hier bewusst vermieden, da er neben der hier gemeinten Bedeutung häufig auch als Bezeichnung für die Gesamtheit der *Speicherseiten* im Adressraum eines Prozesses verwendet wird

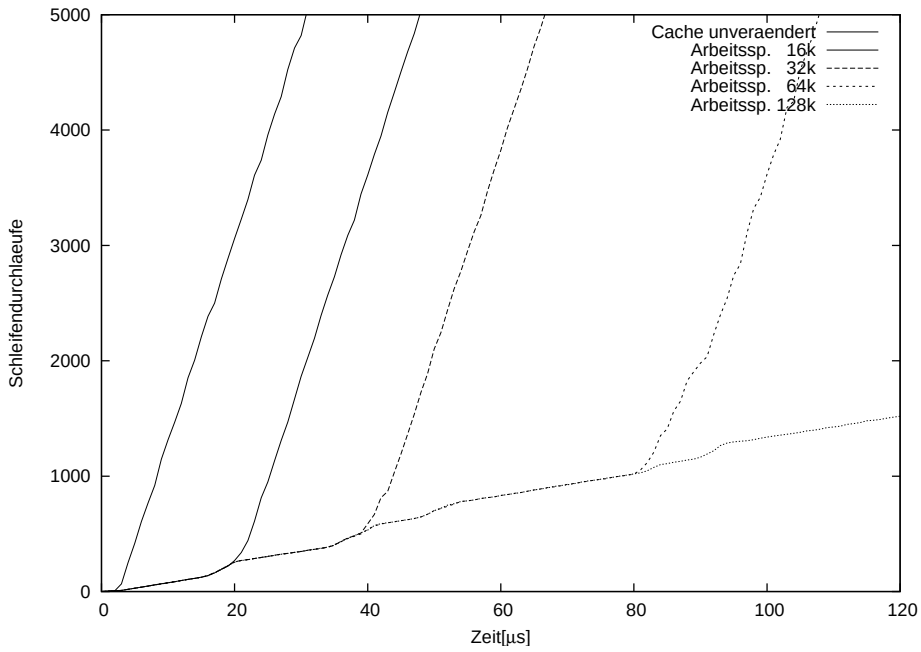

Abbildung 3.11: Programmfortschritt beim Laden des Cache.

leitung nach der Zeit gebildet. Da Abbildung 3.11 den Fortschritt des Programms über der Zeit zeigt, gibt dieses Bild nun die Fortschritts*rate* als Funktion der Zeit an.

Diskussion der Messergebnisse

Alle dargestellten Kurven zeigen das gleiche Muster: Die Fortschittsrate ist zunächst gering, dann steigt sie abrupt an und bleibt dann auf dem Niveau einer höheren, dauerhaften Fortschrittsrate. Diese dauerhafte Fortschrittsrate ist in allen Fällen (außer bei 128k Arbeitsspeichergröße) gleich.

Abbildung 3.13 zeigt die Kurven sämtlicher (insgesamt 20) mit linear aufsteigenden Zugriffsadressen aufgenommenen Messungen für die beiden untersuchten Maschinen mit IA-32-Prozessoren. Dabei wurden alle Fortschrittsratenwerte auf die jeweiligen dauerhaften Fortschrittsraten nach dem sprunghaften Anstieg normiert. Die Ordinatenwerte des Diagramms 3.13 geben somit den Nutzlastfaktor entsprechend Gleichung (3.39) an. Auch in horizontaler Richtung wurden die Kurven normiert: Hierzu wurde der für jede Kurve individuelle Sprungzeitpunkt als Normierungswert herangezogen, sodass alle Kurven in Abbildung 3.13 bei einem Abszissenwert von 1 springen.

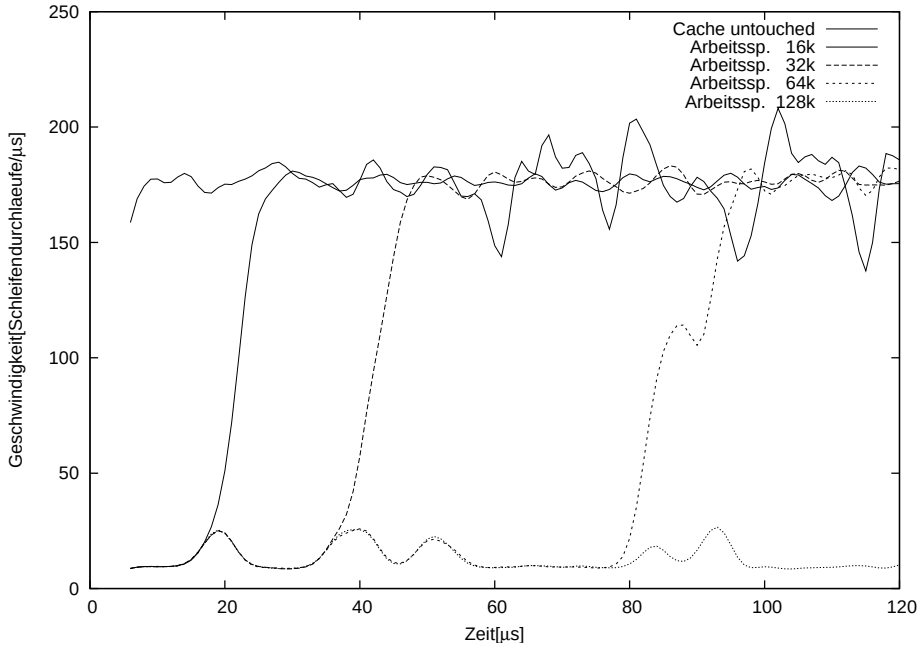

Abbildung 3.12: Fortschrittsrate beim Laden des Cache.

Das Bild unterstreicht die bereits für ausgewählte Beispiele gemachte Feststellung: Obwohl es bei den einzelnen Messkurven große absolute Unterschiede gibt, ist das qualitative Verhalten in allen Fällen gleich: Die Nutzlastfunktion ist zunächst niedrig und bleibt solange auf diesem niedrigen Niveau, bis der gesamte Arbeitsspeicher in den Cache geladen wurde. Ab diesem Punkt springt sie (mehr oder weniger abrupt) auf ein näherungsweise konstantes, höheres Niveau. Dies entspricht dem Verhalten, das durch die „Cache-Fluter" Funktion nach Gleichung (3.41) approximiert werden soll. Der Absolutwert der anfänglichen (niedrigen) Fortschrittsrate variiert zwischen den einzelnen Messungen. Er hängt unter Anderem auch von der Methode ab, nach der der Cache vor der Messung präpariert wurde: wurde der Cache durch Schreibzugriffe gefüllt, so ist die anfängliche Fortschrittsrate niedriger, da dann bei einem Cache-Fehler die im Cache befindlichen Daten in den Hauptspeicher zurückgeschrieben werden müssen. Bei einem Befüllen durch Lesezugriffe ist dies nicht erforderlich, sodass die anfängliche Rate in diesem Fall höher ausfällt.

Tabelle 3.2 zeigt Werte für das Verhältnis zwischen anfänglicher (minimaler) Fortschrittsrate und dauerhafter Fortschrittsrate, f_0, sowie der Zeit t_s bei der der sprunghafte Anstieg zur dauerhaften Fortschrittsrate beobachtet wurde. Diese Werte wurden aus den zuvor gezeigten Messkurven, sowie in analoger Weise aufgenommener Kurven ermittelt, bei denen die Caches vor der Messung nicht

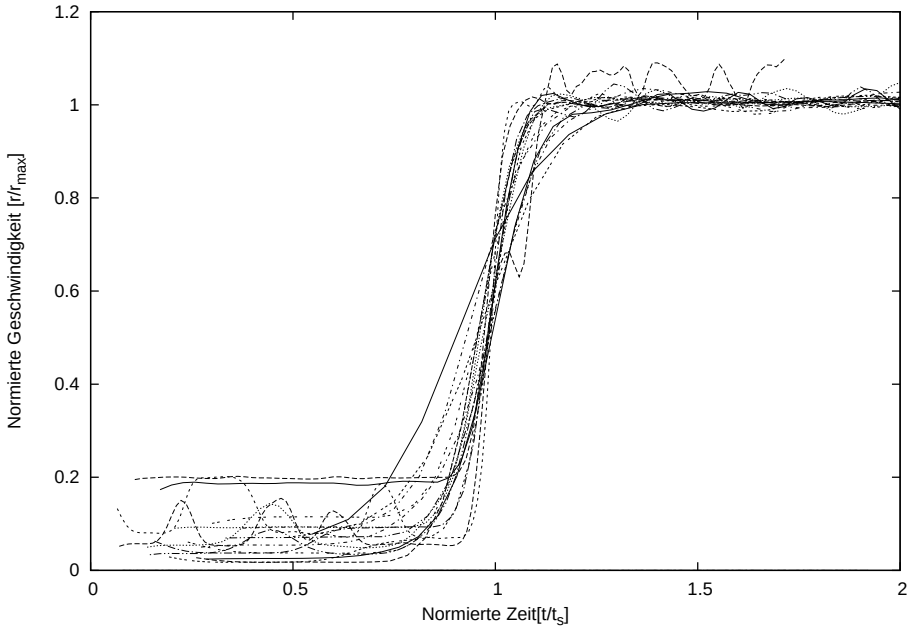

Abbildung 3.13: Normierte Fortschrittsraten (IA-32).

gefüllt, sondern nur gelöscht („invalidiert") wurden. Bei einigen Messungen wurden die referenzierten Speicheradressen zufällig ermittelt, anstatt, wie bisher, ausschließlich linear aufsteigende Adressen zu verwenden.

Es zeigt sich ein eindeutiger Zusammenhang zwischen t_s und der Größe des Arbeitsspeichers. Bei der Maschine mit Celeron-Prozessor ist dieser Zusammenhang erwartungsgemäß proportional, d.h. bei einem doppelt so großen Arbeitsspeicher dauert es doppelt so lang, bis er in den Cache geladen ist. Auch bei der Maschine mit Pentium-M-Prozessor wächst t_s mit der Arbeitsspeichergröße, allerdings ist der Zusammenhang hier nicht proportional. Die dauerhafte Fortschrittsrate ist hier für kleine Arbeitsspeichergrößen deutlich höher. Dieses unerwartete Verhalten hat vermutlich seine Ursache in der zweistufigen Cache-Architektur des IA-32 Prozessors: Der Pentium M hat größere Caches als der Celeron, und, falls der Arbeitsspeicher einer Testsequenz klein genug ist, um vollständig im Level 1 Cache des Prozessors Platz zu finden, so ist die Verarbeitung schneller, als wenn sie nur im größeren Level 2 Cache Platz findet.

Abbildung 3.14 zeigt analog zu Abbildung 3.13 den Verlauf der normierten Fortschrittsraten für den PowerPC MPC5200 Prozessor. Dieser, im vorliegenden Fall mit 400 MHz getaktete Prozessor ist für den Einsatz in eingebetteten Systemen konzipiert. Er besitzt einen einstufigen Cache von 16KB Größe. Dementsprechend wurden bei dieser Messung kleinere Arbeitsspeichergrößen von 2 bis 16KB ge-

Maschine	Adressen	Arbeitssp.	Cache	f_0	t_s
Celeron@2.5GHz	linear	16k	füllen	0.11	$22\mu s$
Celeron@2.5GHz	linear	32k	füllen	0.11	$43\mu s$
Celeron@2.5GHz	linear	64k	füllen	0.09	$85\mu s$
Celeron@2.5GHz	linear	16k	invalidieren	0.17	$12\mu s$
Celeron@2.5GHz	linear	32k	invalidieren	0.18	$20\mu s$
Celeron@2.5GHz	linear	64k	invalidieren	0.20	$35\mu s$
Celeron@2.5GHz	zufällig	16k	füllen	0.26	$30\mu s$
Celeron@2.5GHz	zufällig	32k	füllen	0.36	$38\mu s$
Pentium M@1.5GHz	linear	16k	füllen	0.09	$25\mu s$
Pentium M@1.5GHz	linear	32k	füllen	0.08	$38\mu s$
Pentium M@1.5GHz	linear	64k	füllen	0.12	$91\mu s$
Pentium M@1.5GHz	linear	16k	invalidieren	0.17	$16\mu s$
Pentium M@1.5GHz	linear	32k	invalidieren	0.12	$29\mu s$
Pentium M@1.5GHz	linear	64k	invalidieren	0.21	$55\mu s$
Pentium M@1.5GHz	zufällig	16k	füllen	0.26	$30\mu s$
Pentium M@1.5GHz	zufällig	32k	füllen	0.36	$38\mu s$

Tabelle 3.2: Messergebnisse (IA-32).

wählt. Der Übergang von niedriger zu hoher Fortschrittsrate ist hier deutlich schärfer ausgeprägt, als bei den IA-32 Plattformen.

Tabelle 3.3 gibt die für den MPC5200 aufgenommenen Messwerte an. Hier zeigt sich eindeutig eine proportionale Beziehung zwischen t_s und der Größe des Arbeitsspeichers. Die oben geäußerte Vermutung, die beobachtete nicht-Proportionalität beim Pentium M liege in dessen zweistufiger Cache-Architektur begründet, wird somit bestätigt.

Der anfängliche Nutzlastfaktor f_0 variiert deutlich in Abhängigkeit von der gewählten Methode zur Auswahl der referenzierten Adressen. Dabei ist anzumerken, dass im Falle der zufällig ausgewählten Adressen neben dem Testcode auch noch ein –wenn auch sehr einfacher– Zufallszahlengenerator mit jedem einzelnen Schleifendurchlauf ausgeführt wird, sodass es hier pro Schleifendurchlauf weniger Maschineninstruktionen gibt, die Speicherzugriffe machen und die somit in ihrer Geschwindigkeit durch Cache-Fehler verlangsamt werden. Bei Celeron und MPC5200 gibt es erwartungsgemäß keine Abhängigkeit zwischen der Arbeitsspeichergröße und dem initialen Nutzlastfaktor f_0. Lediglich beim Pentium M scheint eine solche Abhängigkeit zu bestehen: f_0 schwankt zwischen 12% und 21%. Auch dieses Verhalten dürfte im Zusammenspiel zwischen den beiden Cache-Stufen des Pentium M begründet sein.

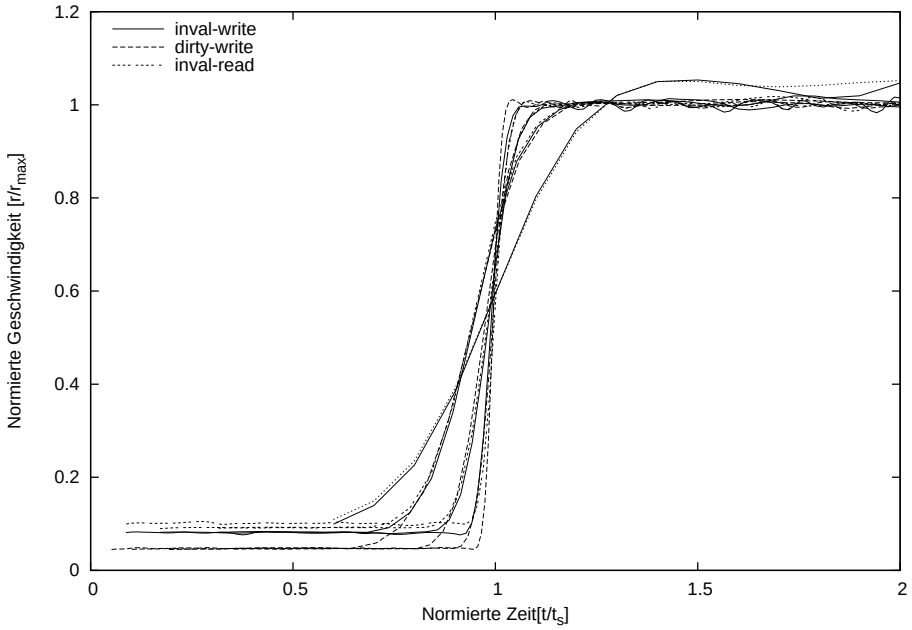

Abbildung 3.14: Normierte Fortschrittsraten (MPC5200).

Trotz dieser leichten Schwankungen kann als Ergebnis festgehalten werden, dass das Verhalten des „Cache-Fluters" mit hinreichender Genauigkeit experimentell nachvollzogen werden konnte. Die Werte für f_0 und t_s in Tabelle 3.2 können somit als Parameter für Gleichung (3.42) angewendet werden, um eine Abschätzung der zu erwartenden Umschaltverluste zu ermitteln

Einfluss der Umschaltverluste auf die Prozessausführung

Die Unterbrechungs- und Umschaltverluste begrenzen die Frequenz, mit der zwischen Prozessen oder auch zwischen virtuellen Maschinen eines Virtual Machine Monitors umgeschaltet werden kann. So arbeitet zum Beispiel die anteilige Prozessorzuordnung (siehe 2.2.8) mit einer endlichen Zeitscheibendauer (dem „Quantum"), als der kleinsten zuteilbaren Menge an Zeit. Das Idealverhalten (d.h. kontinuierliche anteilige Zuteilung) würde bei einem infinitesimalen Quantum erreicht. Dieser Grenzfall ist nicht erreichbar, da mit kleiner werdendem Quantum, d.h. wachsender Umschaltfrequenz, die Umschaltverluste einen zunehmend größeren Teil an Rechenleistung verbrauchen. Um diese Wechselbeziehung zwischen Quantum bzw. Umschaltfrequenz auf der einen und den Verlusten auf der anderen Seite greifbar zu machen, wurde ein C-Programm zur Simulation eines Schedulers zur anteiligen Prozessorzuteilung entwickelt. Dieses Programm (siehe Anhang A)

Maschine	Adressen	Arbeitssp.	Cache	f_0	t_s
MPC52K@400MHz	linear	2k	füllen	0.05	$17\mu s$
MPC52K@400MHz	linear	4k	füllen	0.05	$31\mu s$
MPC52K@400MHz	linear	8k	füllen	0.05	$59\mu s$
MPC52K@400MHz	linear	16k	füllen	0.05	$116\mu s$
MPC52K@400MHz	linear	2k	invalidieren	0.13	$10\mu s$
MPC52K@400MHz	linear	4k	invalidieren	0.09	$19\mu s$
MPC52K@400MHz	linear	8k	invalidieren	0.08	$35\mu s$
MPC52K@400MHz	linear	16k	invalidieren	0.08	$69\mu s$
MPC52K@400MHz	linear (nur lesen)	2k	invalidieren	0.13	$10\mu s$
MPC52K@400MHz	linear (nur lesen)	4k	invalidieren	0.10	$19\mu s$
MPC52K@400MHz	linear (nur lesen)	8k	invalidieren	0.09	$35\mu s$
MPC52K@400MHz	linear (nur lesen)	16k	invalidieren	0.10	$68\mu s$

Tabelle 3.3: Messergebnisse (MPC5200).

berechnet den zeitlichen Verlauf der Zeitzuteilung an eine vorgebbare Anzahl von Prozessen mit wählbaren Gewichten anhand eines von drei zur Wahl stehenden Planungsverfahren zur anteiligen Prozessorzuordnung. Dabei ist es möglich, die durch die Gleichungen (3.42) und (3.44) beschriebenen Umschaltverluste, sowie feste Unterbrechungsverluste in der Simulation mit zu berücksichtigen, d.h. jeder Aufruf des Schedulers hat eine feste Zeitspanne zur Folge, während der kein Prozess fortschreitet, und, falls der Scheduler einen Prozesswechsel vornimmt, arbeitet der neue Prozess entsprechend einer der Gleichungen (3.42) oder (3.44) zunächst langsamer, um dann im Laufe seiner Zeitscheibe nach und nach zu beschleunigen. Damit kann unter Verwendung der experimentell ermittelten Werte für f_0 und t_s der zeitliche Verlauf der einem Prozess (oder einer virtuellen Maschine) nach Abzug der Verluste zur Verfügung stehenden „Netto-Arbeitszeit" berechnet werden.

Abbildung 3.15 zeigt einen solchen Verlauf. Hierbei wurde beispielhaft von einem System mit drei Prozessen ausgegangen, die unter dem in [NVZ01] beschriebenen „Virtual Time Round-Robin" (VTRR) Scheduler arbeiten. Dargestellt ist für einen der drei Prozesse der Verlauf der idealen, proportionalen Rechenzeitzuteilung (siehe „Prop. Zuteilung"), der Zuteilung, wie sie das VTRR-Verfahren durch Vergabe von Zeitscheiben approximiert (siehe „Zeitscheiben, ohne Verluste"), sowie der Zuteilung die nach Abzug von Unterbrechungs- und Umschaltverlusten für die Verarbeitung durch den Prozess übrig bleibt, wobei zur Berechnung der Umschaltverluste wahlweise die Approximationsfunktion $f_{avg}()$ (Gleichung (3.44)) oder $f_{flood}()$ (Gleichung (3.42)) zugrunde gelegt wurde (siehe „Zeitscheiben, f_{avg}", bzw. „Zeitscheiben, f_{flood}"). Beide Funktionen wurden mit den für

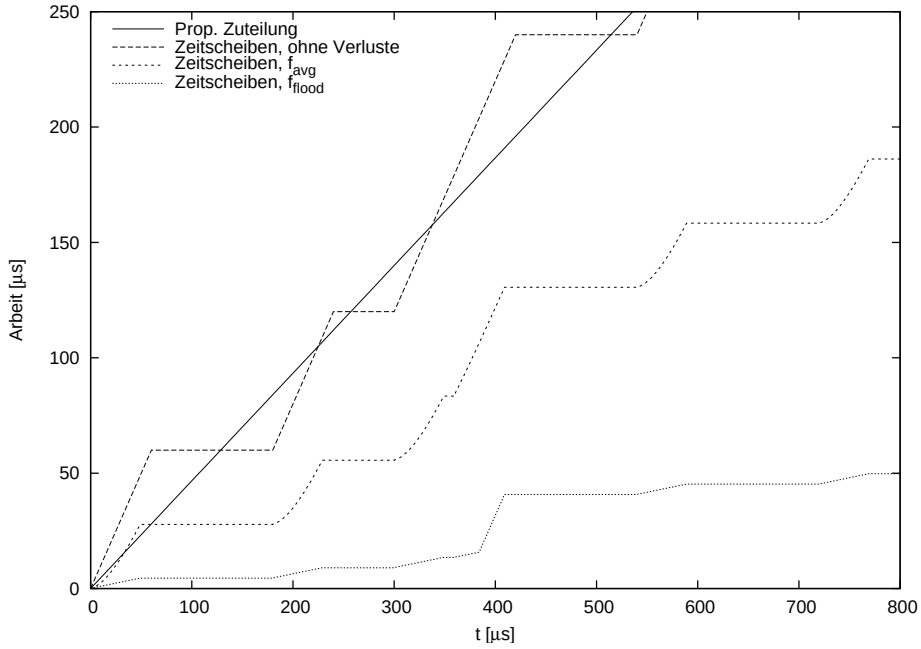

Abbildung 3.15: Effektive Zeitzuteilung unter Berücksichtigung von Umschaltverlusten.

den 2,5 MHz Celeron Prozessor bei 64 KB Arbeitsspeicher gemessenen Werten für f_0 und t_s parametrisiert. Als Unterbrechungsverlust wurde ein realistischer Festwert von 10 μs veranschlagt, die Zeitscheibendauer wurde mit 50 μs extrem kurz angesetzt. Dieser eher unrealistisch kurze Wert wurde gewählt, um den Einfluss der Verluste bereits in den ersten Zeitscheiben deutlich sichtbar zu machen und um den nichtlinearen Verlauf der nutzbaren Rechenzeit im Falle der Verwendung der Approximationsfunktion $f_{avg}()$ (Gleichung (3.44)) zu demonstrieren. Wie sich zeigt, liegt die tatsächlich verfügbare Netto-Zeitzuteilung nach Abzug der Verluste erheblich unter der Brutto-Zuteilung.

Bemerkung: Bei $t = 350\mu s$ findet in diesem Beispiel ein Scheduler-Aufruf statt, der keinen Prozesswechsel zur Folge hat. Daher macht der Prozess hier für 10 μs keinen Fortschritt. Da aber im Anschluss an diese Unterbrechung derselbe Prozess weiterarbeitet, wird in diesem Modell angenommen, dass der Cache in der Zwischenzeit nicht „verschmutzt" wurde, weshalb die Fortschrittsrate nach der Unterbrechung weiter ansteigt.

Diese Simulation wurde nun für verschiedene Zeitscheibenlängen durchgeführt, wobei jeweils die effektiven Verluste ermittelt wurden. Das Ergebnis ist in Abbildung 3.16 wiedergegeben und zeigt den Zusammenhang zwischen der Zeitscheibenlänge, mit der der Scheduler arbeitet, und den sich ergebenden relativen

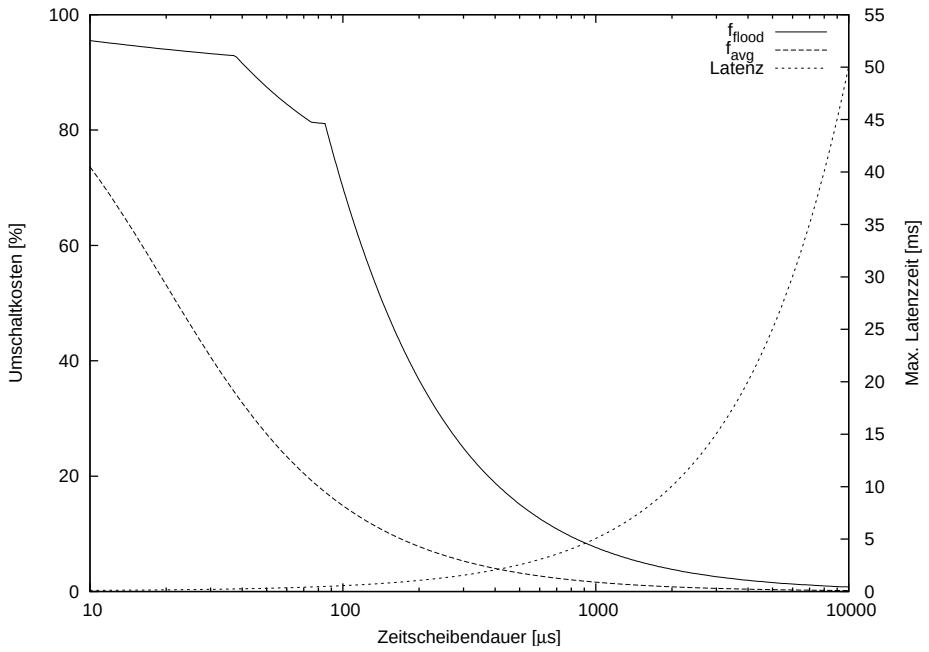

Abbildung 3.16: Umschaltkosten und Latentzzeit als Funktion der Zeitscheibendauer.

Umschaltkosten.

Nach Abbildung 3.16 liegen die Verluste bei Zeitscheibenlängen oberhalb etwa einer Millisekunde im für die meisten Anwendungen tolerierbaren Bereich unterhalb von 10%. Die meisten der heute üblichen Betriebssysteme arbeiten mit Zeitscheibendauern in diesem Bereich, sodass für diese Systeme mit keinen sehr hohen Verlusten durch Prozesswechsel gerechnet werden muss. Unterhalb einer Millisekunde Zeitscheibendauer aber steigen die Verluste drastisch an und erreichen bei 100 μs (bei pessimistischer Schätzung nach Gleichung (3.42)) teilweise über 80%.

Außer den Umschaltkosten zeigt Abbildung 3.16 mit der mit „Latenz" bezeichneten Kurve auch die von den Prozessen beobachteten Maximalwerte der Latenzzeiten der Prozesse. Diese steigt proportional[1] mit der Zeitscheibendauer.

Da die verwendeten Parameter experimentell ermittelt worden sind und da insbesondere Gleichung (3.44) eine eher willkürlich gewählte Approximation realen Verhaltens darstellt, ist das Ergebnis als eine grobe Abschätzung zu verstehen. Es ging hier nicht um eine exakte Vorhersage der im Einzelfall zu erwartenden Umschaltverluste, sondern darum, eine Größenordnung zu identifizieren, oberhalb der

[1] Die Krümmung entsteht durch die logarithmische Skalierung der horizontalen Achse

sich Zeitscheibendauern bei anteiliger Prozessorzuteilung sinnvollerweise bewegen sollten.

Die Ergebnisse zeigen, dass bei Zeitscheibendauern oberhalb etwa einer Millisekunde die Verluste auch unter worst-case Annahmen in einer für die meisten Anwendungen akzeptablen Größenordnung liegen. Gleichzeitig kann festgestellt werden, dass mit der Zeitscheibendauer auch die Latenzzeiten ansteigen. Diese Zeiten haben erheblichen Einfluss auf das Echtzeitverhalten der Prozesse[1]. Abbildung 3.16 liefert somit auch eine Größenordnung für die Verluste, die in Kauf genommen werden müssen, wenn das Echtzeitverhalten durch entsprechend kürzere Zeitscheibendauern verbessert werden soll.

3.3 Betriebssysteme in Virtualisierungsumgebungen

Bei den Gästen einer Virtualisierungsumgebung handelt es sich in der Regel um ganze Betriebssysteme, die mitsamt ihren Anwendungen jeweils in einer virtuellen Maschine arbeiten. Diese Betriebssysteme sind nicht für die Virtualisierung konzipiert, sondern sie erwarten eine reale Hardware, auf der sie arbeiten, bzw. deren Betriebsmittel sie verwalten. Ein Virtual Machine Monitor muss diesen Betriebssystemen eine diesen Anforderungen angemessene Schnittstelle bieten. Dabei sind Ein- und Mehrprozessorsysteme zu unterscheiden.

Wie in 2.3.3 beschrieben, können bei der Vollvirtualisierung Betriebssysteme unverändert in einer virtuellen Maschine arbeiten, während bei der Paravirtualisierung Modifikationen am Code des Betriebssystems (nicht aber am Code der Anwendungen) nötig sind. Dabei werden aber lediglich die sensitiven, nichtprivilegierten Befehle, die die Kriterien von Popek und Goldberg (s. [PG74]) verletzen, durch entsprechende Hypercalls substituiert. Am logischen Aufbau des Betriebssystems wird, ebenso wie (offensichtlich) auch bei der Vollvirtualisierung, nichts verändert. Unabhängig von der Form der Virtualisierung arbeitet ein Gastbetriebssystem also konzeptuell weiterhin genauso, wie es auch auf realer Hardware arbeiten würde.

Eine Eigenschaft realer Rechensystem-Hardware, die großen Einfluss auf die Konstruktion eines Betriebssystems hat, ist die Anzahl der physischen Prozessoren, die verwaltet werden (genauer gesagt: Die Frage, ob es sich dabei um Einen oder Mehrere handelt). Daher wird im Folgenden eine Klassifikation von Betriebssystemen nach diesem Kriterium vorgenommen.

[1] Der Zusammenhang dieser Größen wird in Abschnitt 4.1.3 noch eingehender untersucht.

3.3.1 Klassifikation: Ein- und Mehrprozessorbetriebssysteme

Betriebssysteme bieten als wesentliche Grundfunktionalität Multitasking oder Multiprogramming. Das bedeutet, sie unterstützen das (in der Regel dynamische) Erzeugen einer im Allgemeinen unbegrenzten Anzahl von Prozessen, die sie mit Hilfe ihres Schedulers auf eine endliche Anzahl physischer Prozessoren abbilden. Diese Anzahl physischer Prozessoren tritt auf der Ebene der Anwendungen üblicherweise[1] nicht in Erscheinung. Dennoch ist die Frage, ob mehrere Prozessoren oder nur ein einziger genutzt werden, von großer Bedeutung für die interne Konstruktion eines Betriebssystems. Es muss daher zwischen Ein- und Mehrprozessorbetriebssystemen unterschieden werden (Siehe Abbildung 3.17):

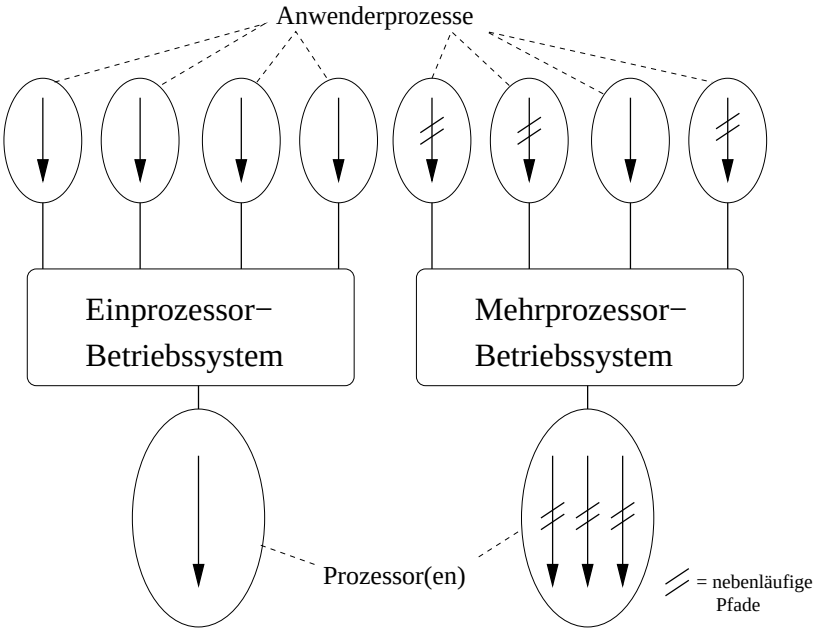

Abbildung 3.17: Ein- und Mehrprozessorbetriebssystem.

Einprozessorbetriebssysteme verwenden nur einen einzigen Prozessor, der reihum, d.h. zeitversetzt verschiedenen Anwenderprozessen zugeteilt wird.

Mehrprozessorbetriebssysteme arbeiten mit mehreren Prozessoren auf Basis eines SMP-Systems mit UMA- oder NUMA-Architektur (vgl. 2.1.1). Die Anwenderprogramme (deren Anzahl in der Regel die der Prozessoren

[1] Es gibt hierzu Ausnahmen, wie das „Cm*"-System, das Ousterhout in [Ous80] beschreibt. Hierauf wird in Kapitel 4 noch eingegangen

übersteigt) werden nach Bedarf verschiedenen Prozessoren zugeordnet, wobei maximal so viele Anwenderprozesse gleichzeitig aktiv sein können, wie Prozessoren zur Verfügung stehen.

Nach dem in 3.1 eingeführten Modell kann ein Einprozessorbetriebssystem zusammen mit seinen Anwenderprozessen durch einen einzigen Stellvertreterprozess beschrieben werden. Bei einem Mehrprozessorbetriebssystem schreitet jeder der Prozessoren entlang eines eigenen Programmpfades fort, d.h. jeder Prozessor stellt einen eigenständigen Teilprozess eines Parallelprozesses dar. Das dieser Situation entsprechende Modell ist also ein paralleler Stellvertreterprozess mit einer festen Anzahl von Teilprozessen. Wie in Abbildung 3.17 angedeutet, können einige der Anwenderprozesse parallel arbeiten, wenn der Scheduler sie verschiedenen Prozessoren zuordnet. Die meisten Mehrprozessorsysteme verbergen diese Tatsache jedoch vor ihren Anwendungen, sodass aus deren Sicht Ein- und Mehrprozessorsysteme nicht voneinander unterscheidbar sind. Die Anwendungen werden entweder als voneinander unabhängige Prozesse oder als grobgranulare Parallelprozesse behandelt, deren Synchronisation über Betriebssystemfunktionen geschieht.

3.3.2 Mechanismen zum wechselseitigen Ausschluss

Im Zuge der Realisierung von Systemfunktionen durchläuft ein Betriebssystem immer wieder kritische Abschnitte, für die sichergestellt werden muss, dass sie zu jeder Zeit nur von einem einzigen Programmpfad durchlaufen werden. In einem Einprozessorbetriebssystem lässt sich eine solche „atomare" Ausführung kritischer Sektionen einfach dadurch realisieren, dass ein Ausführen des Schedulers vorübergehend unterbunden wird: Solange der Scheduler gesperrt ist, kann der Prozessor keinem anderen Prozess zugeteilt werden, womit Betriebsmittelkonflikte ausgeschlossen sind. Bei Mehrprozessorbetriebssystemen genügt dies jedoch nicht: Arbeitet ein konkurrierender Prozess auf einem anderen Prozessor, so sind auch bei gesperrtem Scheduler Konflikte möglich. Um auch Prozesse auf anderen Prozessoren auszuschließen, werden daher in der Regel zusätzlich sogenannte „Spinlocks" verwendet. Hierbei handelt sich um gemeinsame Variablen, die per Konvention ausschließlich mit Hilfe spezieller, unteilbarer Lese-Schreib-Befehle[1] manipuliert werden. Ist die Spinlock-Variable gesetzt, so gilt das Spinlock als belegt und der zugehörige kritische Abschnitt als gesperrt. Wenn ein Prozess auf ein belegtes Spinlock trifft, so wartet er auf dessen Freigabe durch den Prozess, der sich gerade in der kritischen Sektion befindet. Dieses Warten geschieht in

[1] z.B.TAS- (*test-and-set*) oder CAS-Befehl (*compare-and-swap*)

der Regel *aktiv*, d.h. der wartende Prozessor wird *nicht* freigegeben, sondern das Spinlock wird zyklisch mit hoher Frequenz wiederholt geprüft.

3.3.3 Mechanismen zur Synchronisation

Synchronisationen zwischen Prozessen erfordern in einem Einprozessorsystem grundsätzlich ein blockierendes Warten: Wenn ein Prozess auf ein Ergebnis eines anderen Prozesses wartet, so muss er notwendig den Prozessor abgeben, damit der andere Prozess überhaupt fortschreiten und somit zu seinem Ergebnis gelangen kann. Aktives Warten mit Hilfe von Spinlocks kann hier das System zum Stillstand bringen, wenn der wartende Prozess für unbegrenzte Zeit den Prozessor belegt, ohne den der Prozess, auf den er wartet, nicht fortschreiten kann. In einem Mehrprozessorbetriebssystem hingegen kann der andere Prozess unabhängig vom wartenden Prozess fortschreiten, sofern er über einen eigenen Prozessor verfügt. Somit ist ein Blockieren des wartenden Prozesses nicht zwangsläufig. Je nach Dauer der Wartezeit ist es hier sogar effizienter, aktiv zu warten, da der Aufwand zum Blockieren und zum späteren Fortsetzen des Prozesses höher ist (vgl. [Ous80]).

3.3.4 Konsequenzen für die Virtualisierung

Ein- bzw. Mehrprozessorbetriebssysteme müssen also unterschiedliche Mechanismen zum wechselseitigen Ausschluss und zur Prozesssynchronisation verwenden. Während Spinlocks in Mehrprozessorsystemen effizienter sind, führt ihre Verwendung bei Einprozessorsystemen zu Fehlern. Ein- und Mehrprozessor- Betriebssysteme unterscheiden sich somit in ihrem internen Aufbau grundlegend, selbst wenn sie ihren Anwendungen die gleiche Benutzerschnittstelle anbieten.

Arbeitet ein Betriebssystem als Gast in einer virtuellen Maschine, so arbeitet es mit *virtuellen* Prozessoren, die durch den Virtual Machine Monitor bereitgestellt werden. Dabei handelt es sich um Prozesse des Virtual Machine Monitors: Ebenso wie der Scheduler eines Betriebssystem bildet auch der VM-Scheduler des Monitors sie auf physische Prozessoren ab. Dabei bestehen grundsätzlich die gleichen Möglichkeiten wie bei einem Betriebssystem, d.h. virtuelle Prozessoren könnten dynamisch erzeugt werden und ihre Anzahl ist unabhängig von der Anzahl physischer Prozessoren. Allerdings können weder Ein- noch Mehrprozessor-Gastsysteme diese Möglichkeiten sinnvoll nutzen:

- Ein Einprozessor-Gastsystem wird gegenüber dem Virtual Machine Monitor durch einen einzigen Stellvertreterprozess repräsentiert. Dieser Stellver-

treterprozess schreitet entlang eines einzigen Ausführungspfades fort und kann somit nur einen einzigen virtuellen Prozessor nutzen.

- Ein Mehrprozessor-Gastsystem wird durch einen parallelen Stellvertreterprozess mit einer festen Anzahl von Teilprozessen repräsentiert. Er kann somit nur auf einer während seiner Laufzeit konstanten Anzahl von virtuellen Prozessoren arbeiten. Ein dynamisches Erzeugen weiterer virtueller Prozessoren bringt keinen Nutzen. Auch wenn auf Seiten der Anwenderschnittstelle ein Mehrprozessorbetriebssystem nicht von einem Einprozessorbetriebssystem unterscheidbar ist, so stellt doch sein Stellvertreterprozess aus Sicht des Virtual Machine Monitors als mittelgranularer Parallelprozess eine völlig andere Art der Rechenlast dar. Bei realer Hardware werden die Teilprozesse durch die parallel arbeitenden physischen Prozessoren realisiert. Dementsprechend müsste ein Virtual Machine Monitor einem Mehrprozessor-Gastsystem eine dessen Konfiguration entsprechende Anzahl von parallelen virtuellen Prozessoren anbieten. Von den virtuellen Prozessoren, die ein Virtual Machine Monitor an seiner Schnittstelle anbietet, können zu jedem Zeitpunkt aber maximal so viele parallel arbeiten, wie es physische Prozessoren gibt. Die Fähigkeit eines Virtual Machine Monitors, mehr virtuelle Prozessoren anzubieten als physische Prozessoren existieren, impliziert daher, dass einige der virtuellen Prozessoren nicht parallel arbeiten. Werden solche „quasi-parallel" zueinander arbeitenden Prozessoren als Teilprozesse des parallelen Stellvertreterprozesses herangezogen, so verletzt dies Annahmen unter denen das Mehrprozessorsystem konstruiert wurde. Hieraus ergeben sich Probleme, die in 4.3.5 noch eingehend diskutiert werden.

Obwohl also ein Virtual Machine Monitor, ebenso wie ein Betriebssystem, als wesentliche Eigenschaft die Abbildung von Prozessen auf physische Prozessoren bietet, gelten für seine Schnittstelle doch völlig andere Anforderungen: Einem Anwender (d.h. hier: einem Gastbetriebssystem) müssen, abhängig davon ob es sich um ein Ein- oder Mehrprozessorbetriebssystem handelt, ein oder mehrere virtuelle Prozessoren angeboten werden, wobei deren Anzahl begrenzt und während der gesamten Lebensdauer der virtuellen Maschine konstant ist. Alle diese virtuellen Prozessoren sollten stets verschiedenen physischen Prozessoren zugewiesen sein, d.h. sie sollten parallel, nicht quasi-parallel sein.

Kapitel 4

Prozessplanung für virtuelle Maschinen auf eingebetteten Mehrprozessorplattformen

In diesem Kapitel werden die Anforderungen an eine Prozessplanung für ein Mehrprozessorsystem, das als Plattform für die in der Einleitung geforderte Konsolidierung eingebetteter Systeme dienen kann, herausgearbeitet. Die drei Themengebiete der Arbeit, Virtualisierung, Echtzeitverarbeitung und Parallelverarbeitung in Mehrprozessorsystemen werden dazu jeweils paarweise betrachtet, wobei sich Teil-Anforderungen ergeben. Der letzte Abschnitt des Kapitels fasst die ermittelten Anforderungen zusammen und skizziert das Arbeitsprinzip eines Schedulers für Virtual Machine Monitore, der die gestellten Anforderungen erfüllt.

4.1 Echtzeit und Virtualisierung

Im Bereich der Server und der Workstations, dem traditionellen Einsatzgebiet der Virtualisierung, bestehen keine oder allenfalls sehr geringe und sehr „weiche" Echtzeitanforderungen. Das Maß der Dinge ist hier der Mensch als Anwender, dessen Geduld nicht überstrapaziert werden sollte. Die geforderten Antwortzeiten liegen daher üblicherweise in der Größenordnung von Sekunden, und ein gelegentliches Überschreiten dieser Schwelle gilt als unproblematisch, solange es nur hinreichend selten geschieht.

Die Anforderung, solche Nicht-Echtzeitanwendungen zu unterstützen, besteht beim Einsatz von Virtualisierungsumgebungen in eingebetteten Systemen auch

weiterhin, z.B. für Navigationssysteme, Entertainment, etc.. Darüber hinaus ist hier aber auch die Fähigkeit zur Einhaltung harter Zeitbedingungen unerlässlich. Dieser Bedarf stellt eine neue Anforderung für Virtualisierungsumgebungen dar. Er entsteht durch die Interaktion von Prozessen mit einem technischen oder physikalischen System, das ein bestimmtes Zeitverhalten aufweist. Echtzeitprozesse müssen diesem Zeitverhalten gerecht werden, d.h. auf Ereignisse muss rechtzeitig reagiert werden, und die Ergebnisse der Datenverarbeitung müssen rechtzeitig vorliegen. Was hierbei konkret unter „rechtzeitig" zu verstehen ist, hängt von der jeweiligen Aufgabe (d.h. dem technisch/physikalischen System) ab und kann über mehrere Größenordnungen variieren: Typische „schnelle" Anforderungen wie etwa das rechtzeitige Auslösen eines Airbag oder die Bestimmung von Zünd- und Einspritzzeitpunkten bei einer Motorsteuerung bedingen Fristen bis in den Mikrosekundenbereich, bei „langsamen" Systemen wie etwa der Temperaturregelung einer Klimaanlage können sie im Sekunden- oder Minutenbereich liegen.

Eine Virtualisierungsumgebung für eingebettete Systeme muss das gesamte Spektrum von Nicht-Echtzeitanwendungen bis hin zu „harten" Echtzeitanwendungen mit kurzen Antwortzeiten unterstützen können. In diesem Abschnitt soll geklärt werden, welche neuen Funktionalitäten eine Virtualisierungsumgebung dazu ggf. implementieren muss.

Zunächst werden einige existierende Ansätze zur Koexistenz von Echtzeit- und Nicht-Echtzeitanwendungen in einem gemeinsamen System untersucht. Diese verwenden *keine* Virtualisierung, wie sie in 2.3 beschrieben wurde, doch basieren alle –wie auch die Virtualisierung– letztendlich auf Prozesshierarchien (siehe 3.1). Die Hinweise, die sich aus diesen Betrachtungen ergeben, sind deshalb auch auf eine Virtualisierungsumgebung direkt anwendbar.

4.1.1 Existierende Ansätze *ohne* Virtualisierung

Im Folgenden werden drei verschiedene Lösungsansätze zur Koexistenz von Echtzeit- und Nicht-Echtzeitanwendungen betrachtet. Das Ziel aller drei Ansätze ist, den Echtzeitanwendungen ein deterministisches Zeitverhalten zu ermöglichen und dabei gleichzeitig mit Nicht-Echtzeitanwendungen eine möglichst hohe mittlere Verarbeitungsleistung sowie eine gute Prozessorauslastung zu realisieren. Diese beiden Ziele lassen sich nur unter Inkaufnahme von Kompromissen miteinander vereinen. Die hier vorgestellten Ansätze unterscheiden sich in der Wahl dieser Kompromisse.

Slotted Priorities

Bollella stellt in [Bol97] eine als „Slotted Priority Architecture" bezeichnete Architektur vor, bei der der Prozessor zyklisch zwischen einem Echtzeit- und einem Nicht-Echtzeit-Betriebssystem umgeschaltet wird. Dabei haben sowohl das Echtzeit- als auch das Nicht-Echtzeitsystem jeweils feste, als *minor cycle* bezeichnete Zeitscheiben, deren Summe die Zykluszeit, die auch als *major cycle* bezeichnet wird, ergibt. Dieser Ansatz ist zugleich auch ein einfaches Beispiel für die in Abschnitt 3.1.1 beschriebene Zeitpartitionierung. Durch die strikte zeitliche Trennung der beiden „Welten" des Echtzeit- und des Nicht-Echtzeitteils entstehen zwei Subsysteme, die zeitlich voneinander vollständig entkoppelt sind. Über eine „räumliche" Entkopplung der beiden Welten finden sich in [Bol97] nur wenige Aussagen, da die Arbeit sich vornehmlich mit den Aspekten der Echtzeitplanung auseinandersetzt. In jedem Fall gibt es hier getrennte Betriebssystemschnittstellen für Echtzeit- und Nicht-Echtzeit-Anwendungen, zwischen denen beispielsweise über gemeinsame Speicherbereiche kommuniziert werden kann. Eine Priorisierung einzelner Prozesse findet ausschließlich innerhalb einer der beiden Welten statt, d.h. innerhalb jeweils einer der Zeitscheiben (daher der Name der Architektur).

Es handelt sich hier um eine zweistufige Prozesshierarchie, bei der der Scheduler der unteren Ebene statisch, d.h. zeitgesteuert arbeitet. Aufgrund der zeitlichen Trennung der beiden Subsysteme ist es ist hier nicht möglich, im Echtzeitsystem dynamisch anfallende, überschüssige Rechenzeit für die Nicht-Echtzeitverarbeitung zu nutzen. Stattdessen muss jedes Subsystem seine überschüssige Rechenzeit jeweils durch einen eigenen „idle-"Prozess konsumieren. Die zeitliche Entkopplung wird also, wie bei der Zeitpartitionierung üblich, mit einer u.U. schlechten Prozessorauslastung erkauft.

Prioritätssteuerung am Beispiel LynxOS

Eine einfache und weit verbreitete Möglichkeit zur Umwidmung dynamisch anfallender, überschüssiger Rechenzeit ergibt sich bei Anwendung der prioritätenbasierten Prozessorzuteilung (vgl. 2.2.8). Dazu werden die Echtzeitprozesse höher priorisiert, sodass Nicht-Echtzeitprozesse immer nur dann ausgeführt werden, wenn kein Echtzeitprozess rechenwillig ist. Es gibt zahlreiche Beispiele für Implementierungen dieser Technik, von denen hier mit LynxOS ein typischer Vertreter genannt wird.

Bei LynxOS [GL91] unterliegen alle Prozesse einem prioritätsgesteuerten Scheduler. Die Prioritäten sind statisch[1].

[1] Es gibt zwar Systemfunktionen, mit denen Prioritäten zur Laufzeit verändert werden können,

Prozessen mit Echtzeitanforderungen werden bei ihrer Erzeugung höhere Prioritäten zugewiesen, als Prozessen ohne oder mit weniger strengen Echtzeitanforderungen. Die Vergabe der Prioritäten kann dabei zum Beispiel nach dem ratenmonotonen Verfahren (siehe 2.2.7) erfolgen. In der Regel wird für jeden Echtzeitprozess eine individuelle Priorität festgelegt. Nicht-Echtzeitprozesse hingegen erhalten eine gemeinsame, niedrige Priorität. Prozesse mit gleicher Priorität werden vom Scheduler nach dem Zeitscheibenverfahren behandelt, sodass die ihnen gemeinsam zugewiesene Rechenkapazität unter diesen Prozessen aufgeteilt wird. Die Aufteilung ist aber keineswegs gleichmäßig: Prozesse die –z.B. aufgrund von Ein/Ausgabe-Operationen– blockieren, werden benachteiligt, da der bis zum Zeitpunkt der Blockierung noch nicht genutzte Anteil der Zeitscheibe durch die Blockierung verfällt. Das führt dazu, dass Ein/Ausgabe-intensive Prozesse bei gleicher Priorität im Durchschnitt weniger Rechenzeit erhalten, als beispielsweise rechenintensive Prozesse. Ein „faires" Scheduling, wie es für Nicht-Echtzeitprozesse wünschenswert wäre, unterstützt LynxOS nicht.

Obwohl LynxOS intern nur einen einzigen Scheduler besitzt, kann man auch hier mit einer gewissen Berechtigung von einer Prozesshierarchie sprechen: Die erste Ebene der Hierarchie ist die prioritätsbasierte Prozessteuerung, während die zweite Ebene, d.h. der „lokale Scheduler", in dem auf einer Prioritätsstufe angewendeten Zeitscheibenverfahren besteht. Letztere ist bei LynxOS betont einfach ausgeführt, da das System seinen Schwerpunkt auf die Echtzeitverarbeitung legt: Das Zeitscheibenverfahren erlaubt bei Kenntnis der Zeitscheibendauern und Prioritäten anderer Prozesse noch das Nachvollziehen der Prozessabfolge. Daher ist es für ein Echtzeitsystem vorzuziehen. Bei einem komplexeren, auf Nicht-Echtzeitanwendungen zugeschnittenen Verfahren wäre diese Nachvollziehbarkeit (und damit deterministisches Zeitverhalten) nicht unbedingt gegeben.

Prioritätssteuerung am Beispiel RTAI

RTAI [MDP00] ist ein Echtzeitbetriebssystem, das, wie LynxOS, mit einem prioritätsgesteuerten Zeitscheiben-Scheduler arbeitet. Seine Programmierschnittstelle bietet einen für die Echtzeitverarbeitung konzipierten Funktionsumfang. RTAI ist aber nicht eigenständig, sondern es nutzt ein unterlagertes Linux-System als Infrastruktur. Linux arbeitet aus der Sicht des RTAI Systems als „idle-"Prozess auf niedrigster Priorität. Immer dann, wenn keiner der von RTAI verwalteten Echtzeitprozesse Rechenbedarf hat, kommt Linux zur Ausführung. Auf diese Weise wird RTAI den Anforderungen der Echtzeitprozesse gerecht, während die für Nicht-Echtzeitprozesse erforderliche faire Rechenzeitaufteilung durch das unterlagerte

jedoch geschieht dies nur unter Kontrolle von Anwenderprogrammen. Das Betriebssystem selbst nimmt keine eigenständigen Veränderungen der Prioritäten vor.

Linux bewerkstelligt wird. RTAI ist damit ein Beispiel einer zweistufigen Prozesshierarchie (vgl. 3.1): Der Scheduler des RTAI-Systems schaltet als erste Stufe ausschließlich prioritätsbasiert zwischen einer Menge von Echtzeitprozessen um, von denen einer als Stellvertreterprozess das gesamte Linux-System mit seinem lokalen Scheduler kapselt (s. Abbildung 4.1).

Unter den gekapselten Linux-Prozessen findet sich (auf niedrigster Linux-Priorität) auch der Linux „idle-"Prozess, der stets rechenbereit, d.h. gierig ist. Damit ist auch der Stellvertreterprozess des Linux-Subsystems als einziger der vom RTAI-Scheduler gesteuerten Prozesse gierig und konsumiert sämtliche ihm zugewiesene Rechenkapazität. Da er zugleich die niedrigste RTAI-Priorität besitzt, ist diese zugewiesene Rechenkapazität gleich der Summe der von den höher priorisierten Echtzeitprozessen nicht genutzten Anteile an der gesamten Rechenkapazität.

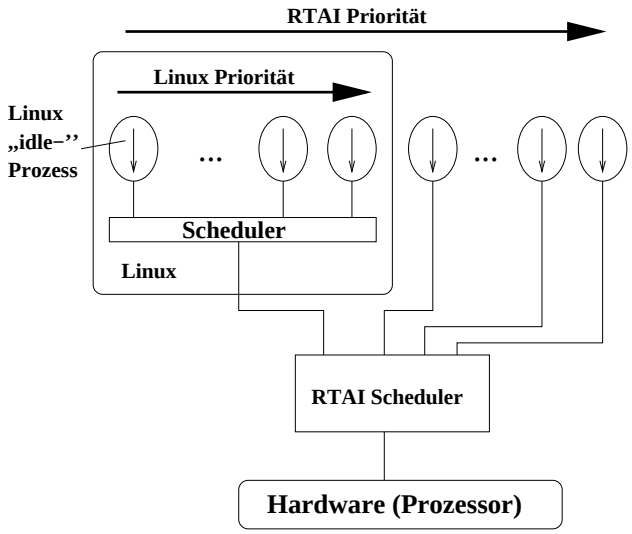

Abbildung 4.1: Scheduler-Hierarchie bei RTAI.

Ein häufiger Fehler bei der Verwendung von RTAI, der aber prinzipiell bei jedem auf statischen Prioritäten basierenden System möglich ist, ist das Erzeugen eines Echtzeitprozesses, der (typischerweise aufgrund von Programmierfehlern) niemals blockiert. Ein solcher ungewollt gieriger Prozess kann aufgrund seiner hohen Priorität sämtliche Prozesse mit niedrigerer Priorität, darunter insbesondere auch das Linux-Subsystem dauerhaft blockieren, sodass unter Anderem auch keine Konsol-Eingaben mehr möglich sind, mit denen der fehlerhafte Prozess ggf. wieder beendet werden könnte.

Der von den hochprioren Prozessen konsumierte Anteil an Rechenkapazität ist nicht beschränkt, bzw. seine Beschränktheit beruht lediglich auf der Annahme

(und dem Vertrauen darauf), dass diese hochprioren Prozesse nicht gierig sind. Eine Überwachung oder gar Beschränkung der durch Echtzeitprozesse konsumierten Rechenzeit ist nicht vorgesehen. Das bedeutet, dass jeder Prozess einer gegebenen Priorität auf das kooperative Verhalten *aller* Prozesse mit höherer Priorität vertrauen muss. Anders ausgedrückt existiert eine „Zwangskopplung" zwischen Priorität und Vertrauenswürdigkeit. Ein Prozess, der, weil er harten Zeitbedingungen unterliegt, eine hohe Priorität erhält, wird dadurch zugleich in die Lage versetzt, Prozesse mit niedrigerer Priorität dauerhaft zu verdrängen. .

Für ein System wie RTAI, bei dem alle Echtzeitprozesse in einem gemeinsamen Adressraum arbeiten[1], sodass sie einander ohnehin uneingeschränkt vertrauen müssen, mag diese Vorgehensweise akzeptabel sein. Zwischen virtuellen Maschinen darf hingegen kein gegenseitiges Vertrauen vorausgesetzt werden. Die in der Einleitung aufgestellte Forderung nach Fehlerlokalität ließe sich sonst nicht erfüllen.

In [HPOS05] wird eine Variante von RTAI vorgestellt, bei der Echtzeitprozessen neben der Priorität zusätzlich noch vom Betriebssystem überwachte Rechenzeitkontingente zugewiesen werden. Überschreitet ein Prozess sein Kontingent, so wird ihm vom Betriebssystem der Prozessor zwangsweise entzogen. Dadurch bleiben die Folgen von Zeitüberschreitungen auf diejenigen Prozesse beschränkt, die sie verursacht haben, ohne dass andere Prozesse dadurch beeinträchtigt werden können.

Fazit

Die betrachteten Beispiele demonstrieren eine grundsätzliche Schwierigkeit bei der Koexistenz von Echtzeit- und Nicht-Echtzeitprozessen. Es ergibt sich ein Zwiespalt zwischen dem erreichbaren Grad an Entkopplung zwischen den beiden Prozessklassen und der Effizienz des Systems:

- Werden Echtzeit- und Nicht-Echtzeitprozesse durch Partitionierung zeitlich entkoppelt, so ist zwar eine sichere Trennung erreichbar, aber gleichzeitig verschlechtert sich auch die Auslastung und damit die Effizienz des Systems.
- Werden sie unterschiedlich hoch priorisiert, so lässt sich zwar eine gute Prozessorauslastung erreichen, aber gleichzeitig ergeben sich Abhängigkeiten zwischen Priorität und Vertrauenswürdigkeit, die mit der in der Einleitung geforderten Fehlerlokalität nicht zu vereinbaren sind.

[1] Es gibt allerdings eine Erweiterung von RTAI namens „LXRT", bei der auch RTAI-Prozesse durch eigene Adressräume gegeneinander geschützt werden können.

Der in [HPOS05] vorgestellte Ansatz einer Kombination aus Zeit- und Prioritätssteuerung zeigt einen brauchbaren Lösungsweg auf: Durch die zeitgesteuerte Begrenzung der von Echtzeitprozessen (bzw. im Falle der Virtualisierung: deren Stellvertreterprozessen) konsumierten Rechenkapazität können diese unabhängig von ihrer Priorität daran gehindert werden, das Zeitverhalten anderer Echtzeitprozesse über das ihnen bei der Planung zugestandene Maß hinaus zu beeinträchtigen. Gleichzeitig erlaubt die Prioritätssteuerung, die bei der Echtzeitverarbeitung anfallenden Überschüsse an Rechenkapazität für die Nicht-Echtzeitverarbeitung dynamisch umzuwidmen.

Ein weiterer Aspekt, der bereits in der Einleitung angesprochen wurde, kann anhand der hier betrachteten Beispiele bestätigt werden: Aus den jeweiligen Anforderungen von Echtzeit- und Nicht-Echtzeitanwendungen ergeben sich widersprüchliche Entwurfskriterien für das zugrunde liegende Betriebssystem. Sollen sowohl Echtzeit- als auch Nicht-Echtzeitanwendungen auf einer gemeinsamen Betriebssystemschnittstelle arbeiten, so müssen entweder auf Seiten der Echtzeitverarbeitung oder –wie am Beispiel LynxOS gezeigt– auf Seiten der Nicht-Echtzeitverarbeitung Kompromisse in Kauf genommen werden. RTAI und Bollellas Ansatz der „Slotted Priorities" bieten dagegen eigene Programmierschnittstellen für Echtzeitanwendungen und überlassen die Nicht-Echtzeitanwendungen einem dafür besser geeigneten Nicht-Echtzeitbetriebssystem. Im Falle von RTAI kommt hierfür Linux zum Einsatz. Die „Welten" von Echtzeit- und Nicht-Echtzeitsystemen sind bei Bollella und –zumindest in der in [HPOS05] vorgestellten Variante– auch bei RTAI voneinander getrennt: Echtzeit- und Nicht-Echtzeitprogramme arbeiten auf eigenen und funktional sehr verschiedenen Programmierschnittstellen. Allerdings hat diese Trennung nur den Charakter einer Empfehlung: Da beide Schnittstellen in einem gemeinsamen Adressraum arbeiten, bestehen zahlreiche Möglichkeiten gegenseitiger Einflussnahme. Zum Beispiel ist jede Echtzeitanwendung technisch in der Lage, Funktionen aus der Nicht-Echtzeitschnittstelle aufzurufen. Das Ergebnis solcher unzulässiger Aufrufe ist in aller Regel undefiniertes Systemverhalten. Auch auf der Nicht-Echtzeit-Seite können Anwendungen leicht das Echtzeitsystem in seiner Funktion beeinträchtigen. Die beiden Schnittstellen sind zwar zeitlich, nicht aber „räumlich" entkoppelt. Würden sie in getrennten Adressräumen existieren, so könnten selbst fehlerbehaftete oder gar böswillige Programme auf einer der beiden Seiten die jeweils andere Seite nicht beeinflussen.

Eine Virtualisierungsumgebung kann diese „räumliche" Trennung bieten. Ob (und wie) sie auch die Forderung nach zeitlicher Entkopplung erfüllen kann, soll im Folgenden weiter untersucht werden.

4.1.2 Echtzeitfähigkeit existierender Virtualisierungsumgebungen

In Abschnitt 3.1.2 wurde bereits auf die grundsätzlichen Probleme bei der Ausführung von Echtzeitprozessen in virtuellen Maschinen hingewiesen. Diese Aussage soll nun durch eine Betrachtung zweier populärer Virtualisierungsumgebungen geprüft werden. Beide wurden in 2.3.7 bereits vorgestellt und als Virtualisierungsumgebungen eingeordnet. Hier soll nun ihr Zeitverhalten betrachtet werden.

VMware ESX Server

VMware ESX Server enthält einen eigenen VM-Scheduler, der die Strategie der Umschaltung zwischen den virtuellen Maschinen definiert. Diese Strategie, die in [VMw05b] beschrieben ist, zielt vornehmlich auf proportionale Verteilung von Rechenkapazitäten ab: Jede virtuelle Maschine erhält einen konfigurierbaren durchschnittlichen *relativen* Anteil der Rechenkapazität. Es finden sich keine Angaben, auf welchen Zeitraum sich die Durchschnittsbildung dabei bezieht. Technisch erforderlich ist eine Stückelung der zugeteilten Zeit in Form von Zeitscheiben, jedoch bietet VMware ESX Server keine Möglichkeit, die absolute Zeitscheibendauer zu beeinflussen. Diese Dauer ist aber von großer Bedeutung für die Latenzzeiten und Fristen, die innerhalb virtueller Maschinen ablaufende Echtzeitprozesse einhalten können. Darüber hinaus wird, falls eine virtuelle Maschine keinen Rechenbedarf hat (z.B. wenn gerade keiner ihrer Prozesse rechenwillig ist), sofort zur nächsten virtuellen Maschine umgeschaltet. Diese im Sinne einer guten Prozessorauslastung an sich vorteilhafte Vorgehensweise hat zur Folge, dass die Zeitpunkte, zu denen eine virtuelle Maschine Rechenkapazität erhält, nicht vorherbestimmt sind und dass sie nicht zuletzt auch durch das Verhalten anderer virtueller Maschinen beeinflusst werden. Sie können daher in der Prozessplanung eines Gastbetriebssystems nicht berücksichtigt werden. Damit bleibt einem Gastbetriebssystem für seine Prozessplanung nur die Möglichkeit, von einer konstanten, anteiligen Prozessorzuteilung auszugehen. Wie in 4.1.3 gezeigt wird, ist auf dieser Basis eine Echtzeitplanung zwar grundsätzlich möglich, die erzielbaren Latenzzeiten und einhaltbaren Fristen sind aber zwangsläufig erheblich schlechter als ohne Virtualisierung.

Fallstudie: Xen

In [BDF+03] wird bezüglich der Strategie, nach der bei Xen die Umschaltung zwischen den virtuellen Maschinen erfolgt, auf [DC99] verwiesen. Der dort beschriebene „BVT"-Scheduler (*„borrowed virtual time"*) zielt auf eine anteilige

Verteilung von Rechenzeit an seine Prozesse, d.h. im Falle von Xen an die virtuellen Maschinen. Als Besonderheit erlaubt es der BVT-Scheduler, dringende Aktionen „vorzuziehen". Xen verwendet diesen Mechanismus, um kurze Interrupt-Latenzzeiten zu erzielen. Ob ein Prozess von dieser Fähigkeit Gebrauch machen kann, hängt dabei unter anderem davon ab, wie oft er dies in der Vergangenheit bereits getan hat, d.h. das Verfahren kann zwar die mittlere Interrupt-Latenzzeit verkürzen, nicht aber die worst-case Zeit. Wenn eine virtuelle Maschine keinen Rechenbedarf hat, gibt sie ihre Zeitscheibe auf, d.h. wie bei VMware ESX Server kann ein Gastbetriebssystem seine Prozessplanung nicht auf das Umschalten zwischen virtuellen Maschinen abstimmen.

Aktuelle Versionen von Xen verwenden anstelle des BVT-Schedulers den sogenannten „Credit-"Scheduler, über den sich nur wenig Literatur findet. In [Chi07] gibt es eine knappe Beschreibung des Verfahrens. Danach ist auch der Credit-Scheduler eine Variante der anteiligen Prozessorzuteilung (siehe 2.2.8). Virtuellen Maschinen können Gewichte zugewiesen werden, anhand derer ihr relativer Anteil an der gesamten zu verteilenden Rechenkapazität bestimmt wird. Optional kann einer virtuellen Maschine ein „Deckelwert" (engl. *cap*) zugewiesen werden, der den (absoluten) maximalen Anteil der zuweisbaren Rechenkapazität festlegt. Der Scheduler arbeitet mit einem festen Quantum von 30 ms.

Um die Eignung von Xen als Plattform für Echtzeitprozesse besser bewerten zu können, wurde folgendes Experiment durchführt:

- Ein Xen System mit zwei virtuellen Maschinen wurde konfiguriert.
- In der ersten virtuellen Maschine („Domain0") arbeitete ein Linux System.
- In der zweiten virtuellen Maschine („DomainU") arbeitete, basierend auf dem zu Xen gehörigen Minimalbetriebssystem „Mini-OS", ein periodischer Echtzeitprozess mit einer konstanten Auslastung von 60%.
- Die Periodendauer des Echtzeitprozesses wurde in Schritten von 5 Millisekunden variiert.
- Das in Domain0 arbeitende Linux war während des gesamten Versuches entweder untätig („idle") oder durch ein Anwenderprogramm (Endlosschleife) voll ausgelastet.
- Gemessen wurde die Latenzzeit des Echtzeitprozesses, d.h. die Differenz zwischen Bereitzeit und Startzeit, $s_i - r_i$. Von diesem Wert wurden über jeweils 10000 Messungen Minimum, Maximum und Durchschnittswert ermittelt.

Abbildungen 4.2 und 4.3 zeigen die Ergebnisse dieses Experiments. Dabei gibt die mit „Grenze" bezeichnete Gerade die Grenze an, oberhalb derer die Summe aus gemessener Latenzzeit und Rechenzeit Δe_i des Echtzeitprozesses größer als die Periodendauer Δp_i ist. Für alle Punkte, die oberhalb dieser Geraden liegen, gilt, dass der Echtzeitprozess erst nach Ende seiner Periode beendet wurde, d.h. in diesem Bereich findet kein fristgerechtes Arbeiten mehr statt.

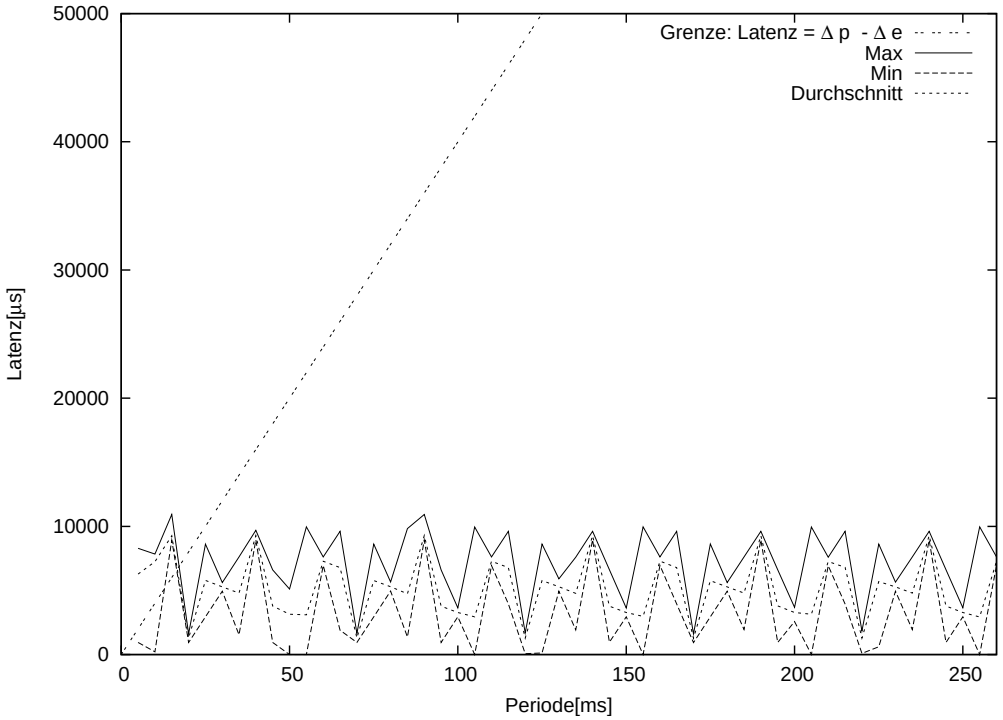

Abbildung 4.2: Latenzzeit eines periodischen Prozesses unter Xen mit Linux in Domain0 im Leerlauf.

Die Ergebnisse zeigen, dass die beobachtete Latenzzeit nach oben begrenzt zu sein scheint[1]. Damit wäre grundsätzlich die Möglichkeit der Echtzeitverarbeitung innerhalb einer virtuellen Maschine vorstellbar. Allerdings liegt die beobachtete Obergrenze mit bis zu 40 Millisekunden etwa um drei Zehnerpotenzen über den bei modernen Echtzeitbetriebssystemen üblichen Werten.

Darüber hinaus zeigt ein Vergleich der Abbildungen 4.2 und 4.3, dass das in Domain0 arbeitende Linux-Betriebssystem erheblichen Einfluss auf das in DomainU arbeitende Echtzeitsystem ausübt.

[1]Es handelt sich hier nur ein Experiment, aus dem keine Garantie über die Begrenztheit der Latenzzeit abgeleitet werden kann.

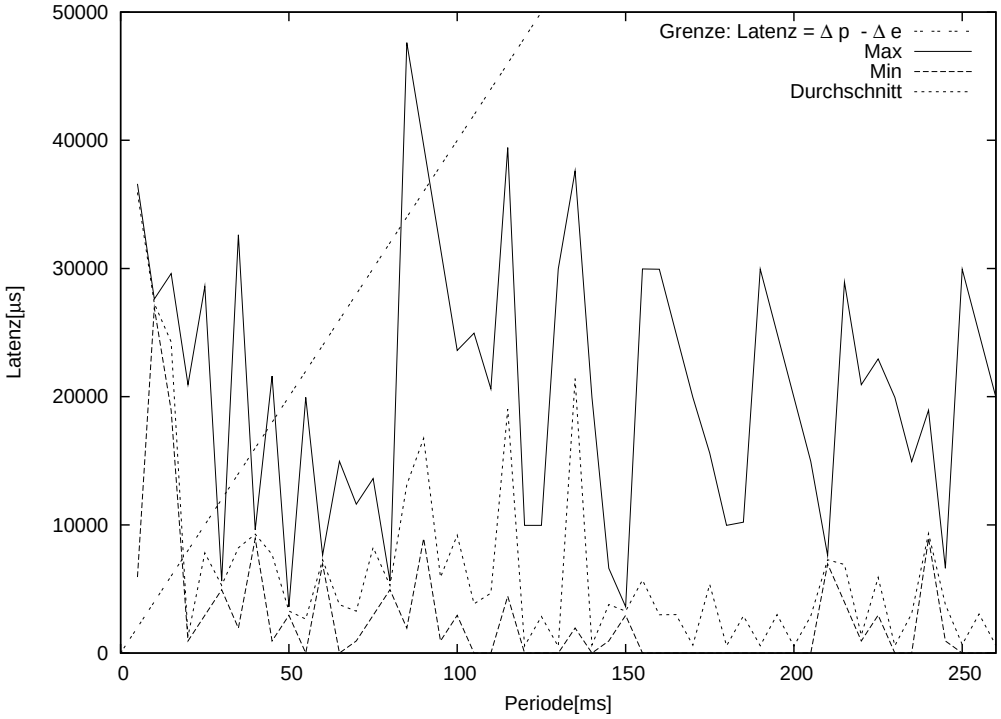

Abbildung 4.3: Latenzzeit eines periodischen Prozesses unter Xen mit Linux in Domain0 unter Volllast.

Hinweis: Diese Messungen wurden mit einem Rechner älterer Bauart (Pentium II @800 MHz) durchgeführt. Eine spätere Wiederholung des Experiments mit einer aktuelleren Maschine zeigte deutlich kürzere Latenzzeiten im ersten Fall, d.h. wenn keine andere virtuelle Maschine aktiv ist (vgl. 6.2.2). Im zweiten Fall (d.h. gleichzeitige Auslastung durch eine zweite virtuelle Maschine) zeigte sich allerdings wieder das gleiche Resultat, sodass die Feststellung der zeitlichen Abhängigkeit der virtuellen Maschinen gültig bleibt..

Fazit

Die beiden Virtualisierungsumgebungen, die hier stellvertretend für eine Vielzahl existierender Produkte für den Server- und Workstation-Markt untersucht wurden, sind offensichtlich nicht für den Einsatz in eingebetteten Systemen entwickelt worden. Bezeichnenderweise findet sich in der Liste der unter Xen lauffähigen Gastbetriebssysteme keines mit ausgewiesenen Echtzeiteigenschaften. In erster Linie wird Linux verwendet, aber auch andere Betriebssysteme (NetBSD, FreeBSD, Plan9, Windows XP) werden bzw. wurden zeitweise unterstützt. Die

eingesetzten Planungsverfahren sind als Varianten der anteiligen Prozessorzuteilung (siehe 2.2.8) für die Echtzeitverarbeitung immerhin bedingt brauchbar (wenn auch bei VMware eine Kontrolle über die Zeitintervalle, über die die konfigurierbaren Prozessoranteile zugesichert werden, fehlt). Die experimentellen Ergebnisse von Xen bestätigen dies: Die beobachteten Latenzzeiten sind offenbar nach oben beschränkt, doch liegen sie um mehrere Größenordnungen über Werten, wie sie von heutigen Echtzeitbetriebssystemen ohne Virtualisierung erreicht werden. Von einem menschlicher Benutzer werden solche Latenzzeiten kaum wahrgenommen, und es gibt durchaus Bereiche der Echtzeitverarbeitung, für die sie akzeptabel sind. Für die meisten Anwendungsbereiche, in der Avionik, der Automation oder in Kraftfahrzeugen können derart hohe Latenzzeiten jedoch in der Regel nicht hingenommen werden.

Eine weitere Beobachtung ist, dass zwar eine sichere räumliche Trennung der Betriebsmittel verschiedener virtueller Maschinen gegeben ist, dass aber keine zeitliche Entkopplung besteht: Getrennte virtuelle Maschinen beeinflussen sich gegenseitig in ihrem Zeitverhalten. Echtzeitanwendungen müssen neben der korrekten Funktion auch ein korrektes und deterministisches Zeitverhalten garantieren können. Die Einflussnahme einer virtuellen Maschine auf das Zeitverhalten einer anderen stellt demnach für Echtzeitanwendungen eine Möglichkeit zur unkontrollierten Fehlerausbreitung über die Grenzen der virtuellen Maschinen hinweg dar. Dies steht im Widerspruch zu der eingangs aufgestellten Forderung nach Fehlerlokalität.

4.1.3 Echtzeitfähigkeit bei anteiliger Zuteilung

In Abschnitt 4.1.2 wurden zwei existierende Virtualisierungsumgebungen empirisch auf ihre Eignung zur Echtzeitverarbeitung untersucht. Wie bei den meisten Systemen dieser Art basieren deren VM-Scheduler auf dem Prinzip der anteiligen Prozessorzuordnung (siehe 2.2.8). Dieses entspricht der Zielsetzung einer Virtualisierungsumgebung: Den in den virtuellen Maschinen ablaufenden Prozessen soll die Illusion gegeben werden, ihnen stünden permanent die Betriebsmittel einer realen Maschine zur Verfügung, obwohl es sich in Wirklichkeit nur um Teilmengen der Betriebsmittel einer entsprechend größeren realen Maschine handelt. Angewendet auf das Betriebsmittel „Prozessor" bedeutet dies, dass jeder virtuellen Maschine ein prozentualer Anteil an der gesamten Rechenleistung der realen Maschine zuteil wird. Idealerweise wäre dieser Anteil kontinuierlich über die Zeit verteilt, sodass die einer virtuellen Maschine zugeteilte Ausführungszeit –wie durch die strichpunktierten Linien in Abbildung 4.4 angedeutet– linear anwachsen würde. Dies entspräche der idealisierten Form der anteiligen Prozessorzuordnung (vgl. 2.2.8). Tatsächlich aber kann der VM-Scheduler die Prozessoren immer nur

zwischen den Kontexten verschiedener virtueller Maschinen umschalten, wobei ein Prozessor jeweils nur für die Dauer einer endlichen Zeitscheibe einer bestimmten virtuellen Maschine zugeteilt bleibt. Dadurch ergibt sich –wie mit den durchgezogenen Linien in Abbildung 4.4 angedeutet– ein „rampenartiger" Verlauf der Ausführungszeit einer virtuellen Maschine, der den idealisierten, linearen Verlauf approximiert. Die Genauigkeit dieser Approximation steigt mit kürzer werdenden Zeitscheibendauern.

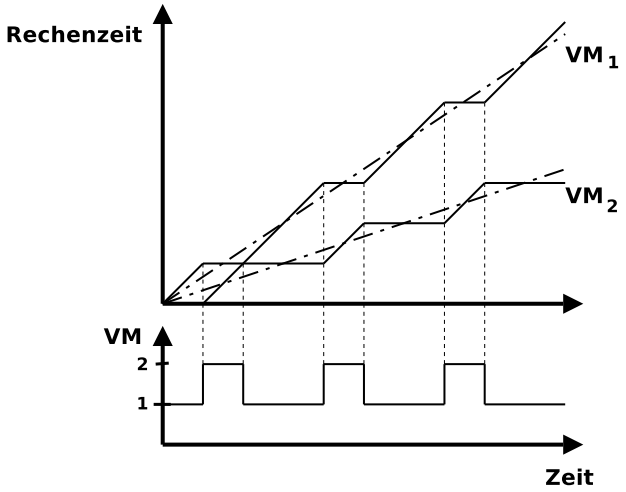

Abbildung 4.4: Idealisierter und realer Verlauf der Rechenzeit virtueller Maschinen.

Idealerweise sollte der VM-Scheduler bei gleichbleibender relativer Verteilung der Rechenkapazitäten auf die virtuellen Maschinen infinitesimal kleine Zeitscheiben verwenden, er sollte also möglichst häufig umschalten. Einer solchen Erhöhung der Umschaltfrequenz sind jedoch durch die mit den Umschaltungen verbundenen Verluste (vgl. 3.2.2) Grenzen gesetzt. Die meisten der existierenden Virtualisierungsumgebungen vernachlässigen die Differenz zwischen Approximation und idealisiertem Verhalten. Die zeitlichen Fehler, die dadurch entstehen, sind für die im Server- und Workstation-Bereich üblichen Nicht-Echtzeitanwendungen durchaus akzeptabel. Für viele Echtzeitanwendungen im Umfeld eingebetteter Systeme sind sie jedoch den empirischen Beobachtungen am Beispiel von Xen zufolge (siehe 4.1.2) zu groß. In diesem Abschnitt sollen anhand eines einfachen Modells die Grenzen der Echtzeitfähigkeit von Virtualisierungsumgebungen aufgezeigt werden, deren VM-Scheduler nach der Methode der anteiligen Zuteilung arbeitet. Dazu wird das Zeitverhalten eines Prozesses betrachtet, wenn er

1. auf einer realen (d.h. nicht-virtuellen) Maschine, oder

2. innerhalb der Zeitscheibe einer virtuellen Maschine

arbeitet. Abbildung 4.5 zeigt die beiden Fälle. Der Prozess werde zu dem Zeitpunkten r_j rechenbereit und infolgedessen zu den (späteren) Zeitpunkten s_j rechnend. Es wird hier angenommen, dass es für den Scheduler, dem der Prozess unterliegt, keine Gründe gibt, den Start des Prozesses zu verzögern[1]. Dann ist im ersten Fall (d.h. ohne Virtualisierung) die Verzögerung zwischen Bereit- und Startzeit ausschließlich durch die Unterbrechungsverluste, d.h. durch die Unterbrechungszeit Δe_{br} bestimmt (siehe 3.2.2).

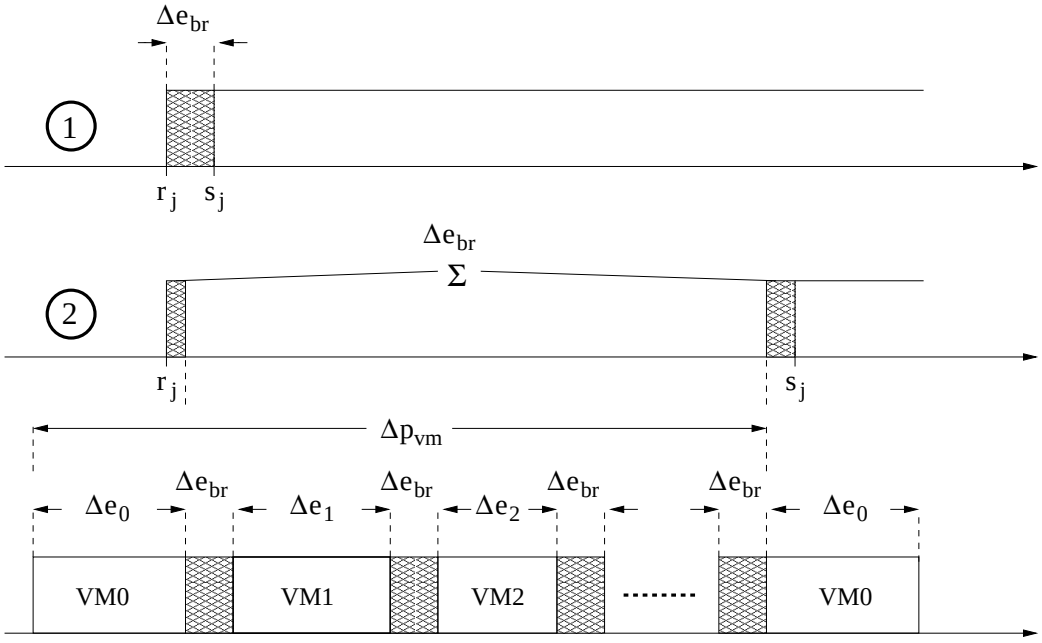

Abbildung 4.5: Prozessumschaltung ohne (1) und mit (2) Virtualisierung.

Im Folgenden wird angenommen, dass diese Zeit eine im Wesentlichen durch die Eigenschaften der verwendeten Hardware vorgegebene Konstante ist:

$$\forall j \in \mathbb{N} : s_j - r_j = \Delta e_{br} = const \tag{4.1}$$

Für den zweiten Fall, d.h. unter einer Virtualisierungsumgebung wird davon ausgegangen, dass der Prozess immer nur während der Zeitscheibe der ihm zugeordneten virtuellen Maschine arbeiten kann. Die Dauer dieser Zeitscheibe sei Δe_0, und sie werde periodisch mit der Periodendauer Δp_{vm} wiederholt. Abbildung 4.5

[1] D.h. es gibt zum Beispiel keinen höher priorisierten Prozess, der gerade rechnet.

zeigt die Zeitscheiben der virtuellen Maschinen. Unter günstigen Bedingungen ist es möglich, dass Bereit- und Startzeit in derselben Zeitscheibe des VM-Schedulers liegen. In diesem Fall wäre die Latenzzeit ebenfalls gleich der Unterbrechungszeit Δe_{br}. Allgemein kann aber nicht davon ausgegangen werden, dass die Aktivitäten innerhalb einer virtuellen Maschine mit denen des VM-Schedulers korreliert sind. Unter ungünstigen Bedingungen (und nur diese sind bei der Echtzeitplanung zu berücksichtigen) kann, wie in Abbildung 4.5 dargestellt, die Bereitzeit so kurz vor dem Ende der aktuellen Zeitscheibe der virtuellen Maschine liegen, dass der Prozess erst in der nächsten Zeitscheibe gestartet wird. Die Latenzzeit erhöht sich in diesem Fall um die Zeitscheibendauern aller virtueller Maschinen außer derjenigen, in der der Prozess selbst arbeitet, zuzüglich der Unterbrechungszeiten zwischen diesen virtuellen Maschinen. Die Umschaltung zwischen virtuellen Maschinen durch einen Virtual Machine Monitor und die Umschaltung zwischen zwei Prozessen durch ein Betriebssystem sind, technisch gesehen, der gleiche Vorgang: Prozessorregister und Adressraumkonfigurationen müssen gerettet/wiederhergestellt werden, und ein Scheduler muss den nächsten zu aktivierenden Prozess, bzw. die nächste zu aktivierende virtuelle Maschine auswählen. Daher kann angenommen werden, dass die Kosten der Umschaltung zwischen zwei virtuellen Maschinen gleich den Kosten der Umschaltung zwischen Prozessen, also Δe_{br}, ist. Gibt es insgesamt N ($N \geq 2$) virtuelle Maschinen mit den jeweiligen Zeitscheibendauern Δe_i, so ergibt sich für den ungünstigen Fall die Latenzzeit zu (vgl. Abbildung 4.5):

$$s_j - r_j = \Delta e_{br} + N \cdot \Delta e_{br} + \sum_{i=1}^{N-1} \Delta e_i \tag{4.2}$$

$$= \Delta e_{br} - \Delta e_0 + N \cdot \Delta e_{br} + \sum_{i=0}^{N-1} \Delta e_i \tag{4.3}$$

$$= \Delta e_{br} - \Delta e_0 + \Delta p_{vm} \tag{4.4}$$

Das Verhältnis R der Latenzzeiten im ersten und zweiten Fall, d.h. der Faktor, um den sich die Latenzzeit in Folge der Virtualisierung erhöht, ist damit:

$$R = \frac{\Delta e_{br} - \Delta e_0 + \Delta p_{vm}}{\Delta e_{br}} = 1 + \frac{\Delta p_{vm}}{\Delta e_{br}} - \frac{\Delta e_0}{\Delta e_{br}}$$

Mit den Auslastungen durch die virtuellen Maschinen $U_i = \frac{\Delta e_i}{\Delta p_{vm}}$ und der Auslastung durch Unterbrechungsverluste $U_{sw} = N \cdot \frac{\Delta e_{br}}{\Delta p_{vm}}$ wird daraus:

$$R = 1 + \frac{N \cdot (1 - U_0)}{U_{sw}} \tag{4.5}$$

Das Verhältnis der Latenzzeiten hängt also von der Anzahl der virtuellen Maschinen, den Unterbrechungsverlusten und der Kapazität der virtuellen Maschine ab, in der der Echtzeitprozess läuft. Abbildung 4.6 zeigt diese Funktion für 3 bzw. 5 virtuelle Maschinen.

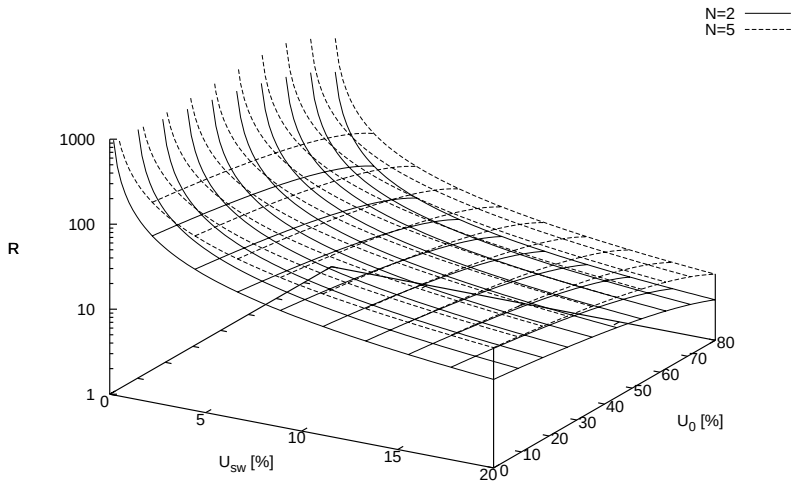

Abbildung 4.6: Erhöhung der Latenzzeit eines Prozesses durch Virtualisierung in Abhängigkeit von Unterbrechungsverlusten und Kapazität der betrachteten virtuellen Maschine für verschiedene Anzahlen von virtuellen Maschinen.

Wird vereinfachend angenommen, dass die Auslastungen, und damit zugleich die Zeitscheiben aller virtueller Maschinen gleich sind[1], d.h.: $\forall i, j \in \mathbb{N} : \Delta e_i = \Delta e_j = \Delta e_{vm}$, so kann das Verhältnis der Latenzzeiten allein durch die Anzahl virtueller Maschinen und die Unterbrechungsverluste ausgedrückt werden:

Aus Gleichung (4.5) wird mit $U_0 = U_{vm}$:

$$\begin{aligned} R &= 1 + \frac{N \cdot (1 - U_{vm})}{U_{sw}} \\ &= 1 + \frac{N}{U_{sw}} - \frac{N \cdot U_{vm}}{U_{sw}} \end{aligned}$$

[1]Dieser Sonderfall wird auch als *processor sharing* bezeichnet (vgl. 2.2.8)

$$= 1 + \frac{N}{U_{sw}} - \frac{\Delta e_{vm}}{\Delta e_{sw}} \tag{4.6}$$

Die Periodendauer ist gleich der Summe der Zeitscheiben plus der Summe der Unterbrechungszeiten. Zugleich ist $U_{sw} = N \cdot \frac{\Delta e_{br}}{\Delta p_{vm}}$ (s.o.):

$$\Delta p_{vm} = N \cdot (\Delta e_{br} + \Delta e_{vm}) = N \cdot \frac{\Delta e_{sw}}{U_{sw}}$$
$$\Rightarrow \frac{\Delta e_{vm}}{\Delta e_{sw}} = \frac{1}{U_{sw}} - 1$$

Dies in (4.6) eingesetzt ergibt:

$$R = 2 + \frac{N-1}{U_{sw}} \tag{4.7}$$

D.h. unter der Annahme gleicher Zeitscheibendauern bzw. Kapazitäten der virtuellen Maschinen ist der Faktor, um den sich die Latenzzeit in Folge der Virtualisierung erhöht, ausschließlich von der Anzahl virtueller Maschinen und den akzeptierten Unterbrechungsverlusten abhängig. Abbildung 4.7 zeigt den Verlauf der Funktion für verschiedene N.

Um ein realistisches Beispiel zu geben: Bei einem System mit zwei virtuellen Maschinen und einem akzeptierten Verlust durch Kontextwechsel von 1% muss mit einer Erhöhung der Latenzzeit um einen Faktor 102 gerechnet werden.

Bei dieser Betrachtung wurden nur die in 3.2.2 beschriebenen Unterbrechungsverluste berücksichtigt. Folglich wurde nur der Einfluss der Virtualisierung auf Latenzzeiten betrachtet. Für eine Betrachtung der Antwortzeiten eines Echtzeitprozesses müssten auch die Umschaltverluste berücksichtigt werden, was zweifellos zu einem noch schlechteren Ergebnis führen wird.

Fazit

Die Virtualisierung mit anteiliger Prozessorzuordnung (bzw. „Processor Sharing") führt zu einer Erhöhung der Latenzzeiten und damit zu einer Verschlechterung des Echtzeitverhaltens von Prozessen. Die Höhe der Verzögerung ist begrenzt: Sie steht in direktem Zusammenhang mit der Anzahl virtueller Maschinen und deren Zeitscheibendauern. Aufgrund dieser Begrenztheit ist die Echtzeitverarbeitung für Prozesse, die in einer solchen Umgebung arbeiten, grundsätzlich möglich, solange die einzuhaltenden Fristen groß gegenüber den Zeitscheibendauern der virtuellen Maschinen sind. Eine Verkürzung dieser Zeitscheibendauern ist aufgrund

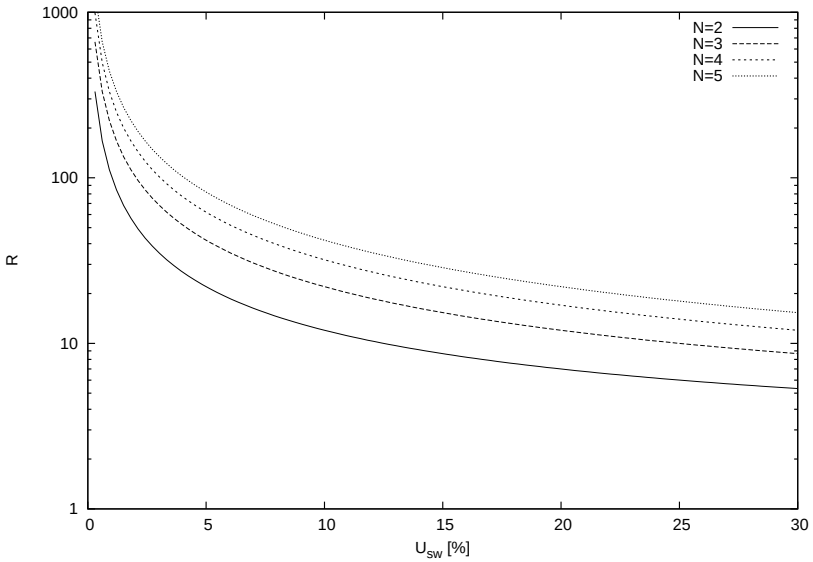

Abbildung 4.7: Erhöhung der Latenzzeit eines Prozesses durch Virtualisierung in Abhängigkeit von Unterbrechungsverlusten für verschiedene Anzahlen von virtuellen Maschinen.

dadurch wachsender Umschaltverluste nur begrenzt möglich. Somit existiert eine Untergrenze für die einhaltbaren Fristen, die für viele praktische Einsatzgebiete zu hoch ist. Dieses Ergebnis deckt sich qualitativ mit den Messungen an Xen in Abschnitt 4.1.2.

4.1.4 Klassifikation der Rechenlasten

Bei der angestrebten Konsolidierung für eingebettete Systeme müssen Anwendungen mit sehr verschiedenen Zeitanforderungen in einem System zusammengeführt werden. Das abzudeckende Spektrum reicht von Nicht-Echtzeitprozessen ohne jegliche Zeitanforderung[1] bis hin zu Echtzeitprozessen mit harten Zeitschranken im Mikrosekundenbereich. Die anteilige Prozessorzuordnung, die üblicherweise in Virtual Machine Monitoren als VM-Scheduler verwendet wird, ist nur für einen Teil dieser Anwendungen geeignet: Wie gezeigt, scheitert das Verfahren, wenn Ausführungszeiten und Fristen in der gleichen Größenordnung wie die Zuteilungseinheit des Schedulers (oder sogar darunter) liegen.

[1] Nach dem in 3.1.4 Gesagten sind Prozesse streng ohne jegliche Zeitanforderung nicht wirklich praxisrelevant: Tatsächlich wird auch von einem Nicht-Echtzeitprozess ein gewisser Mindestfortschritt pro Zeiteinheit vorausgesetzt, was letztendlich bereits eine –wenn auch sehr „weiche"– Echtzeitanforderung darstellt.

Im Folgenden wird zwischen Echtzeit- und Nicht-Echtzeitrechenlasten unterschieden. Dabei sind mit Echtzeitrechenlasten diejenigen Aufgaben gemeint, die mit der anteiligen Prozessorzuordnung nicht bedient werden können. Für sie müssen andere Planungsmethoden gefunden werden. Als Nicht-Echtzeitrechenlasten werden umgekehrt diejenigen Aufgaben bezeichnet, deren Zeitanforderungen nach den Überlegungen aus Abschnitt 4.1.2 mit anteiliger Zuordnung zufriedenzustellen sind. In der Einleitung (Abschnitt: Schnittstellenvielfalt) wurde bereits erwähnt, dass Echtzeit- und Nicht-Echtzeitprogramme zum Teil sehr unterschiedliche Betriebssystemfunktionalitäten erfordern. Die in Abschnitt 4.1.1 betrachteten Beispiele bestätigen diese Aussage. Die Virtualisierung bietet hier die Chance, mehrere Betriebssystemschnittstellen gleichzeitig in einem gemeinsamen Rechensystem zu realisieren, sodass jeder Klasse von Anwendungen ihre eigene, spezialisierte Schnittstelle angeboten werden kann. Aus Sicht des VM-Schedulers stellen die verschiedenen Anwendungsklassen damit verschiedene Klassen von Gastsystemen dar, denen auf unterschiedliche, an die jeweilige Klasse angepasste Weise Rechenkapazitäten zuzuweisen sind.

Echtzeit-Rechenlasten

Echtzeitverarbeitung stellt für Virtualisierungsumgebungen eine neue Anforderung dar. Der Bedarf entsteht durch die Interaktion von Anwendungen mit einem technischen oder physikalischen System, das ein bestimmtes Zeitverhalten aufweist. Ebenfalls durch die Anwendung vorgegeben ist die Entscheidung, ob ein Echtzeitprozess zeit- oder ereignisgesteuert arbeitet (vgl. 2.2.6). Während beispielsweise eine Temperaturregelung oder eine Motorsteuerung eher „proaktiv", d.h. „aus sich heraus arbeitend", und damit zeitgesteuert sind, wäre der Not-Aus-Schalter einer Maschine oder ein Airbag eher „reaktiv", d.h. durch ein äußeres Signal ausgelöst und damit ereignisgesteuert.

Zeitgesteuerte Prozesse arbeiten nach einem statischen, in der Regel periodischen Zeitplan. Sie werden als unterbrechbare periodische Prozesse durch ihre Periodendauer und ihre Ausführungszeit charakterisiert. Ereignisgesteuerte Prozesse müssen sporadisch sein, damit sie überhaupt planbar sind (siehe 2.2.1), d.h. es kann eine kleinstmögliche Periodendauer und eine obere Schranke der Auslastung angegeben werden. Die durch einen solchen Prozess verursachte Prozessorauslastung ist damit ebenfalls beschränkt. Im Extremfall, d.h. bei maximaler Auslastung und minimaler Periodendauer wird ein solcher sporadischer Prozess zu einem periodischen Prozess, der wiederum durch Periodendauer und Ausführungszeit beschrieben werden kann. Dieser Extremfall wird der Planung zugrunde gelegt.

Eine Echtzeit-Rechenlast kann somit als eine Menge unterbrechbarer periodischer Prozesse beschrieben werden. Diese Prozessmenge ist in ihrer Gesamtheit nicht

gierig, d.h. die durch sie verursachte Auslastung ist beschränkt. Nach Abschnitt 3.1.3 kann eine solche Prozessmenge, wenn sie nach dem EDF- oder dem RMS-Verfahren geplant ist, als Ganzes durch einen einzigen unterbrechbaren periodischen Stellvertreterprozess dargestellt werden. Dabei können die charakteristischen Prozessdaten Periodendauer Δp_{sv} und Ausführungszeit Δe_{sv} des Stellvertreterprozesses aus den Prozessdaten der in der Menge enthaltenen Prozesse ermittelt werden. Im Falle der Planung nach RMS ist die Periodendauer Δp_{sv} gleich der kleinsten unter den Periodendauern aller Prozesse der Menge. Bei der Planung nach EDF gibt es in der Regel mehrere mögliche Werte der Periodendauer, die teils unterhalb, teils oberhalb der kleinsten Periodendauer der Prozessmenge liegen. In jedem Fall liegt die Periodendauer des Stellvertreterprozesses in der Größenordnung der in der Menge enthaltenen Prozesse und ist unabhängig von der Zuteilungseinheit des Schedulers.

Aus Sicht des VM-Schedulers wird eine Menge von unterbrechbaren periodischen Prozessen mitsamt deren zugrunde liegendem (Gast-)Betriebssystem als ein solcher Stellvertreterprozess gesehen. Mehrere virtuelle Maschinen, in denen Gastsysteme mit nach EDF oder RMS eingeplanten Prozessen arbeiten, können demnach vom VM-Scheduler ihrerseits wie „gewöhnliche" unterbrechbare periodische Prozesse (z.B. nach RMS oder EDF) eingeplant werden. Findet das Planungsverfahren einen brauchbaren Plan, so ist damit auch für alle Prozesse innerhalb der Gastsysteme eine fristgerechte Ausführung sichergestellt.

Mit $S = \{1, ..n\}$ als Menge der Stellvertreterprozesse und Δe^i_{sv} und Δp^i_{sv} als deren Ausführungszeiten und Periodendauern gilt nach RMS (vgl. 2.2.7):

S ist planbar, wenn

$$U(S) = \sum_{i=1}^{n} \frac{\Delta e^i_{sv}}{\Delta p^i_{sv}} \leq n \cdot \left(2^{\frac{1}{n}} - 1\right) \tag{4.8}$$

bzw. für $n \to \infty$:

$$U(S) \leq ln(2) \tag{4.9}$$

Um den Plan nach RMS durchzusetzen ist ein prioritätsgesteuerter VM-Scheduler mit unterbrechenden Prioritäten erforderlich (vgl. 2.2.8). Den verschiedenen virtuellen Maschinen werden dann Prioritäten derart zugeteilt, dass gilt (vgl. 2.2.7):

$$\Delta p^i_{sv} > \Delta p^j_{sv} \Longleftrightarrow prio(i) < prio(j) \tag{4.10}$$

Die Prioritätszuteilung ist fest, d.h. der VM-Scheduler braucht nur statische Prioritäten zu unterstützen. Wie immer beim Planen nach monotonen Raten kommt

es zu Planungslücken, d.h. zu Zeitintervallen, während derer keine der virtuellen Maschinen, die dem prioritätsgesteuerten Scheduler unterstehen, rechenwillig ist. Diese ungenutzten Rechenkapazitäten sind jedoch nicht „verloren": sie können für die Ausführung von Nicht-Echtzeitrechenlasten verwendet werden (s.u.).

Für das Planen nach EDF gilt: S ist planbar, wenn

$$U(S) = \sum_{i=1}^{n} \frac{\Delta e_{sv}^{i}}{\Delta p_{sv}^{i}} \leq 1 \tag{4.11}$$

Beim Planen nach EDF muss jeweils diejenige virtuelle Maschine ausgeführt werden, die aktuell die kürzeste Frist hat. Falls der Plan mit Hilfe von Prioritäten umgesetzt werden soll, muss der VM-Scheduler dazu die Prioritätszuordnung zur Laufzeit dynamisch verändern können. Anders als bei RMS kann hier die Auslastung bis zu 100% betragen, d.h. es muss nicht notwendigerweise Planungslücken geben. Dies bezieht sich allerdings nur auf die Ebene des VM-Schedulers. Innerhalb der Stellvertreterprozesse, d.h. auf der Ebene des lokalen Schedulers können dennoch Lücken bestehen. Dies gilt in jedem Fall, wenn der lokale Scheduler nach RMS arbeitet, aber auch für EDF sind nach Abschnitt 3.1.3 Planungslücken auf Ebene des lokalen Schedulers nicht immer vermeidbar.

Nicht-Echtzeit-Rechenlasten

Ein Betriebssystem für Nicht-Echtzeitprozesse ist bestrebt, die *durchschnittliche* Bearbeitungszeit von Aufträgen möglichst kurz zu halten. Es verfolgt daher die Strategie, die pro Zeiteinheit verrichtete Rechenarbeit zu maximieren. Dieses Ziel wird erreicht, indem das Betriebsmittel „Prozessor" vollständig ausgelastet wird: Jedem Prozessor ist möglichst zu jedem Zeitpunkt ein Prozess zuzuteilen, d.h. das verwendete Zuteilungsverfahren ist arbeitserhaltend (siehe 2.2.8). Die Prozesse sind unterbrechbare, endliche oder nicht-endliche Nicht-Echtzeitprozesse (vgl. 2.2.2), für die in der Regel vorab keine Planungsdaten vorliegen. Damit kann die Prozessorzuteilung hier nur dynamisch erfolgen. Übersteigt dabei die Anzahl rechenwilliger Prozesse die Anzahl verfügbarer Prozessoren, so versucht das System, alle Prozesse gleichermaßen fair zu behandeln, d.h. jeder Prozess erhält einen Anteil an der verfügbaren Rechenleistung. Gibt es nicht genügend rechenwillige Prozesse, um alle Prozessoren zu beschäftigen, so wird der überschüssige Anteil an Rechenleistung durch Ausführen von „idle-"Prozessen konsumiert.

Nach Abschnitt 3.1.3 lässt sich ein Nicht-Echtzeitbetriebssystem, das eine Menge von Menge von Nicht-Echtzeitprozessen enthält, durch einen Stellvertreterprozess beschreiben. Da in Gestalt des „idle-"Prozesses immer mindestens ein nicht-

endlicher, gieriger Prozess in der Prozessmenge enthalten ist, ist dieser Stellvertreterprozess ebenfalls gierig. Ein Virtual Machine Monitor, der mehrere solcher Nicht-Echtzeitbetriebssysteme als Gastsysteme unterstützen soll, muss die zu ihrer Ausführung insgesamt zur Verfügung stehende Rechenkapazität fair unter ihnen aufteilen, d.h. er muss jedem Nicht-Echtzeitsystem einen –möglicherweise konfigurierbaren– proportionalen Anteil an der gesamten Rechenleistung zuweisen. Ein geeignetes Zuteilungsverfahren hierzu ist die anteilige Zuordnung (s. 2.2.8).

Sei $N = \{1, .., n\}$ eine Menge von Stellvertreterprozessen, die jeweils eine Menge von Nicht-Echtzeitprozessen kapseln. Jeder dieser Stellvertreterprozesse ist durch eine minimale Prozessorauslastung U^i_{min} charakterisiert, die ihm vom VM-Scheduler durchschnittlich gestattet werden muss. Unter der Annahme, dass die Zuteilungseinheit (das „Quantum") des VM-Schedulers klein gegenüber den betrachteten Zeiteinheiten ist[1], kann mit guter Näherung vom idealisierten Modell der kontinuierlichen anteiligen Zuordnung ausgegangen werden. Mit dem Gewicht w_i, das den relativen Anteil eines Stellvertreterprozesses i an der insgesamt verfügbaren Rechenkapazität angibt, ist die gesamte Prozessorauslastung, die von der Menge N beansprucht wird:

$$U_{min} = \frac{1}{\sum_{i=1}^{n} w_i} \cdot \sum_{i=1}^{n} w_i \cdot U^i_{min} \tag{4.12}$$

Der minimale Bedarf an Rechenzeit für die Gesamtheit aller Nicht-Echtzeitsysteme ist damit:

$$req_{min}(t) = U_{min} \cdot t = \frac{t}{\sum_{i=1}^{n} w_i} \cdot \sum_{i=1}^{n} w_i \cdot U^i_{min} \tag{4.13}$$

In einer Virtualisierungsumgebung kann der Bedarf aus drei verschiedenen Quellen gedeckt werden:

1. Durch *Zuweisen eines festen, zyklisch wiederholten Zeitfensters*: Ebenso wie in Abschnitt 3.1.3 für nach RMS oder EDF geplante Echtzeitprozessmengen gezeigt, wird periodisch in jedem Zeitintervall der Dauer Δp_{sv} eine kumulierte Rechenzeit Δe_{sv} zugeordnet. Dieser Anteil kann gegenüber dem Scheduler durch einen unterbrechbaren periodischen Prozess beschrieben

[1] Dies ist das Kriterium, anhand dessen diese Rechenlasten als Nicht-Echtzeitlasten qualifiziert werden

werden. Da er bei der Prozessplanung reserviert werden muss, bezeichnen wir ihn als statischen Anteil. Analog zu Gleichung (3.20) in Abschnitt 3.1.4 ist die auf diese Weise zugewiesene Rechenzeit zum Zeitpunkt t:

$$alloc_{stat}(t) = \left\lfloor \frac{t}{\Delta p_{sv}} \right\rfloor \cdot \Delta e_{sv} + \max(0, t \bmod \Delta p_{sv} - \Delta p_{sv} + \Delta e_{sv})$$

und die Auslastung ist entsprechend:

$$U_{stat} = \frac{\Delta e_{sv}}{\Delta p_{sv}}$$

Die Periodendauer Δp_{sv} ist dabei über einen weiten Bereich frei wählbar: Einerseits sollte sie im Interesse geringer Umschaltverluste möglichst groß sein, andererseits geben die oben erwähnten, in der Praxis auch bei Nicht-Echtzeitanwendungen bestehenden Zeitanforderungen eine (wenn auch „weiche") Obergrenze vor. Im Unterschied zu Echtzeitprozessen muss hier die Begrenzung der pro Periode zur Verfügung gestellten Ausführungszeit vom globalen Scheduler durchgesetzt werden, da der Prozess selbst gierig ist: Nachdem die kumulierte Ausführungszeit innerhalb der Periode den Wert Δe_{sv} erreicht hat, wird der Prozess für den Rest der Periodendauer Δp_{sv} unterbrochen. Diese Vorgehensweise entspricht dem „polling Server" von Buttazzo ([But97], S.111), wobei hier jedoch als Besonderheit der Prozess gierig ist, d.h. er ist immer rechenbereit. Ein „Verfall" der zugesicherten Kapazität, wie er bei Buttazzo für den Fall vorgesehen ist, dass der unter dem Server arbeitende aperiodische Prozess zu Beginn seiner Periode nicht rechenwillig ist, kommt deshalb hier nicht vor.

2. Aus den *Planungslücken der Echtzeitverarbeitung*: Wie in Abschnitt 3.1.4 gezeigt, entstehen bei der Kapselung von nach RMS oder EDF geplanten Prozessmengen in der Regel Planungslücken auf der Ebene der lokalen Scheduler. Dies äußert sich darin, dass einer virtuellen Maschine vom VM-Scheduler eine größere Prozessorauslastung gestattet werden muss, als die darin arbeitenden Prozesse in der Summe tatsächlich benötigen. In 3.1.4 wurden Gleichungen zur Berechnung der notwendigen Zuweisung sowie des Verbrauchs durch die enthaltene Prozessmenge angegeben. Die Differenz zwischen diesen beiden Auslastungen ist somit für nach RMS oder EDF geplante Prozessmengen ebenfalls berechenbar. Sie gibt die durchschnittlich durch Planungslücken ungenutzt bleibende Rechenkapazität an.

Die in einem System insgesamt anfallende Differenz ist bei Kenntnis aller gekapselten Echtzeit-Prozessmengen sowie der von ihnen verwendeten lokalen Planungsverfahren berechenbar. Für eine nach RMS geplante Prozessmenge P_1 gilt beispielsweise nach Gleichung (3.16):

$$\Delta U_1 = U_{sv} - U(P_1) = 2 \cdot \left(1 - e^{-U(P_1)}\right) - U(P_1) \tag{4.14}$$

und für eine nach EDF geplante Prozessmenge P_2 gilt nach Gleichung (3.29)

$$\Delta U_2 = U_{alloc} - U_{req} = \frac{\Delta e_{sv}}{\Delta p_{sv}} - U(P_2) \tag{4.15}$$

wobei die Werte der Parameter Δe_{sv} und Δp_{sv} auch durch die Eigenschaften der vorgegebenen Prozessmenge P_2 festgelegt werden.

Auch auf der Ebene des (globalen) VM-Schedulers können Planungslücken entstehen. Wird dort mit festen Prioritäten nach dem RMS-Verfahren geplant, so gilt nach Gleichung (4.8):

$$\Delta U_0 = 1 - U(S) \geq 1 - n \cdot \left(2^{\frac{1}{n}} - 1\right) \tag{4.16}$$

bzw. für $n \to \infty$:

$$\Delta U_0 \geq 1 - ln(2) \approx 0,3069 \tag{4.17}$$

Insgesamt stehen durch Planungslücken bei n Echtzeitprozessmengen $P_1, ...P_n$ Auslastungsbeiträge in Höhe von

$$U_{gap} = \sum_{i=0}^{n} \Delta U_i$$

zur Verfügung, wobei die Einzelbeiträge ΔU_i, je nach lokalem bzw. globalem Planungsverfahren, nach einer der drei angegebenen Gleichungen (4.16), (4.14) oder (4.15) zu berechnen sind. Diese aus Planungslücken entstehende Rechenkapazität kann (unter Vernachlässigung von Umschaltverlusten) für die Ausführung von Nicht-Echtzeitprozessen dynamisch umgewidmet werden. Um die in einem Intervall $[0, t]$ verfügbare ungenutzte Rechenzeit zu ermitteln, wären alle in diesem Intervall auftretenden Planungslücken aufzusummieren, was in der Regel kaum praktikabel ist. Immerhin

kann für nach EDF geplante Prozessmengen mit Hilfe der Gleichungen (3.22), (3.20) und (3.21) aus Abschnitt 3.1.4 ein entsprechender Ausdruck angegeben werden:

$$\begin{aligned} alloc_{gap}^{EDF}(t) &= slack(t) \\ &= \left\lfloor \frac{t}{\Delta p_{sv}} \right\rfloor \cdot \Delta e_{sv} + \max(0, t \bmod \Delta p_{sv} - \Delta p_{sv} + \Delta e_{sv}) \\ &\quad - \sum_{i=1}^{n} \left\lfloor \frac{t}{\Delta p_i} \right\rfloor \Delta e_i \end{aligned}$$

Um die gesamte zu einem Zeitpunkt t für Nicht-Echtzeitsysteme verfügbare ungenutzte Zeit zu ermitteln, müsste ein solcher Ausdruck für jede virtuelle Maschine, die Echtzeitlasten enthält, berechnet und aufsummiert werden. Mit der vergleichsweise einfach zu berechnenden Auslastung U_{gap} darf streng genommen nur gearbeitet werden, wenn die betrachteten Zeiträume Vielfache der Periodendauer sind, auf die sich U_{gap} bezieht. In diesem Fall gilt einfach:

$$alloc_{gap}(t) = U_{gap} \cdot t$$

Die betreffende Periodendauer ist durch das kleinste gemeinsame Vielfache der Periodendauern aller beteiligten Echtzeitprozesse des Systems gegeben. Dieser Wert kann potenziell sehr groß werden, sodass dieser Beitrag zur Zuteilung an Nicht-Echtzeitprozesse zwar gesichert verfügbar ist, aber in einer Planung nur dann berücksichtigt werden darf, wenn die Nicht-Echtzeitprozesse einen entsprechend großen zeitlichen Spielraum besitzen.

3. Aus *nicht genutzten Ausführungszeiten*: Die in der Planung veranschlagten Ausführungszeiten der Echtzeitprozesse werden anhand von „worst-case"-Annahmen bemessen, damit sie unter allen denkbaren Betriebsbedingungen ausreichend sind. Der tatsächliche Rechenzeitbedarf ist in der Mehrzahl der Fälle wesentlich geringer, d.h. in der Regel liegt die Abschlusszeit c_i^j der Ausführung eines Echtzeitprozesses weit vor der zugehörigen Frist d_i^j (siehe Abbildung 4.8).

   Hinzu kommt, dass sporadische Prozesse eingeplant werden, indem man sie als periodische Prozesse mit maximaler Wiederholfrequenz auffasst. Eine solche Lastsituation kann in der Praxis zwar tatsächlich eintreten, ist aber im

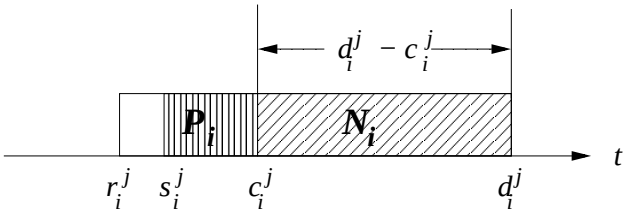

Abbildung 4.8: Überschüssige Rechenzeit wird für Nicht-Echtzeitprozesse genutzt.

Allgemeinen sehr unwahrscheinlich. Damit entfällt eine große Anzahl eingeplanter Ausführungen des betreffenden Prozesses, was in Abbildung 4.8 bedeuten würde, dass $c_i^j = s_i^j$ ist. Dies kann als Grenzfall der nur teilweisen Nutzung der Ausführungszeit aufgefasst werden. Während der ungenutzten Zeitspanne $d_i^j - c_i^j$ können Nicht-Echtzeitprozesse bzw. deren Stellvertreter N_i ausgeführt werden. Die Summe der durch einen Prozess $i \in \{1, ..n\}$ innerhalb einer Zeitspanne von m Perioden der Dauer Δp_i anfallenden, ungenutzten Rechenzeiten ist:

$$\Delta e_{res}^i = \sum_{j=1}^{m} d_i^j - c_i^j$$

Über die Höhe dieses Beitrages sind in der Praxis allenfalls statistische Aussagen möglich. Besteht also die Anforderung an einen VM-Scheduler, eine bestimmte minimale Rechenkapazität für Nicht-Echtzeitprozesse *gesichert* bereitzustellen, so darf dieser Anteil dabei nicht berücksichtigt werden. Ist es hingegen ausreichend, die Rechenkapazität nur im statistischen Mittel zuzusichern, so kann mit Wahrscheinlichkeiten argumentiert werden. Die durchschnittlich pro Periode ungenutzte Zeit ist:

$$\Delta e_{res}^i = \frac{\Delta e_{res}^i}{m} = \frac{1}{m} \cdot \sum_{j=1}^{m} d_i^j - c_i^j$$

Dieser Anteil an der eingeplanten Ausführungszeit liegt zwischen 0 und der „worst-case" Ausführungszeit, Δe_i:

$$0 \leq \Delta e_{res}^i \leq \Delta e_i$$

mit $q_i := \frac{\Delta e^i_{res}}{\Delta e_i}$, $q_i \in \mathbb{R}^+, 0 \leq q_i \leq 1$ als statistisches Mittel der von Prozess i nicht in Anspruch genommenen Rechenzeit gilt für den ungenutzt verbleibenden Anteil der Zuweisung:

$$U_{res} = \sum_{i=1}^{n} \frac{\Delta e^i_{res}}{\Delta p_i} = \sum_{i=1}^{n} q_i \cdot \frac{\Delta e_i}{\Delta p_i}$$

Je nach den Eigenschaften der Prozessmenge kann u.U. auch ein durchschnittlicher Anteil der ungenutzten Zuweisung angegeben werden:

$$\bar{q} := \frac{1}{n} \cdot \sum_{i=1}^{n} q_i$$

Dann gilt:

$$U_{res} = \bar{q} \cdot \sum_{i=1}^{n} \frac{\Delta e_i}{\Delta p_i} = \bar{q} \cdot U(P)$$

Waren bereits über das Anfallen dieser ungenutzten Rechenzeiten nur statistische Aussagen möglich, so trifft dies erst recht auf ihren zeitlichen Verlauf zu. Es wird angenommen, dass diese ungenutzten Rechenzeiten gleichmäßig verteilt anfallen. Unter dieser Annahme gilt

$$alloc_{res}(t) = U_{res} \cdot t = t \cdot \sum_{i=1}^{n} q_i \cdot \frac{\Delta e_i}{\Delta p_i}$$

Die insgesamt zur Ausführung von Nicht-Echtzeitanwendungen verfügbare Rechenzeit ist die Summe der drei Beiträge:

$$alloc(t) = alloc_{stat}(t) + alloc_{gap}(t) + alloc_{res}(t)$$

Von den drei Beiträgen zur Zuweisung ist nur die statische Zuweisung, $alloc_{stat}(t)$ jederzeit gesichert gegeben. Der durch Planungslücken entstehende Anteil $alloc_{gap}(t)$ ist allgemein nur für sehr große t, bzw. für Vielfache des kleinsten gemeinsamen

Vielfachen sämtlicher Periodendauern des Systems gesichert (wenn auch bei vielen praktischen Systemen eine große Wahrscheinlichkeit besteht, dass Planungslücken in etwa gleichmäßig verteilt sind), und der Anteil $alloc_{res}(t)$ ist ohnehin nur statistisch erfassbar, d.h. im ungünstigsten Fall kann er völlig verschwinden.

Soll also der minimalen Rechenzeitbedarf U_{min} (nach Gleichung (4.12)) zu jeder Zeit t gesichert sein, so kann dazu nur der statische Anteil der Zuweisung herangezogen werden. Für alle t muss gelten:

$$t \cdot U_{min} \leq \left\lfloor \frac{t}{\Delta p_{sv}} \right\rfloor \cdot \Delta e_{sv} + \max(0, t \bmod \Delta p_{sv} - \Delta p_{sv} + \Delta e_{sv})$$

beziehungsweise, da $\forall t \geq 0$ gilt:

$$\left\lfloor \frac{t}{\Delta p_{sv}} \right\rfloor \cdot \Delta e_{sv} + \max(0, t \bmod \Delta p_{sv} - \Delta p_{sv} + \Delta e_{sv}) \leq t \cdot \frac{\Delta e_{sv}}{\Delta p_{sv}}$$

muss gelten:

$$U_{min} \leq \frac{\Delta e_{sv}}{\Delta p_{sv}} \tag{4.18}$$

Zur ständigen Deckung dieses Minimalbedarfs muss der VM-Scheduler einen „polling Server", das heißt ein periodisches Zeitfenster bereitstellen, während dessen Nicht-Echtzeitprozesse ausgeführt werden. Dazu kann einem Stellvertreterprozess N_i periodisch in festen Zeitabständen Δp_{sv} jeweils für eine feste Ausführungszeit Δe_{sv} ein Prozessor zugeteilt werden (siehe Abbildung 4.9).

Über einen hinreichend langen Zeitraum betrachtet erhöht sich die Zuweisung an die Nicht-Echtzeitprozesse zusätzlich um den durch Planungslücken der Echtzeitplanung bedingten Beitrag $alloc_{gap}(t)$ sowie im statistischen Mittel um den Beitrag $alloc_{res}(t)$.

Die verschiedenen Beiträge zur für die Nicht-Echtzeitprozesse verfügbaren Rechenkapazität fallen systemglobal an: Es gibt per se keinen einzelnen Nicht-Echtzeitprozess, dem sie zuzuordnen wären. Vielmehr kommt dem VM-Scheduler die Aufgabe zu, diese teils statisch vorgegebenen, teils dynamisch freiwerdenden Kapazitäten anteilig auf alle Nicht-Echtzeitprozesse zu verteilen. Da die freien Rechenzeiten aus Sicht der Echtzeitplanung überflüssig sind, betrifft die Strategie, nach der ihre Verteilung vorgenommen wird, die Echtzeitverarbeitung nicht

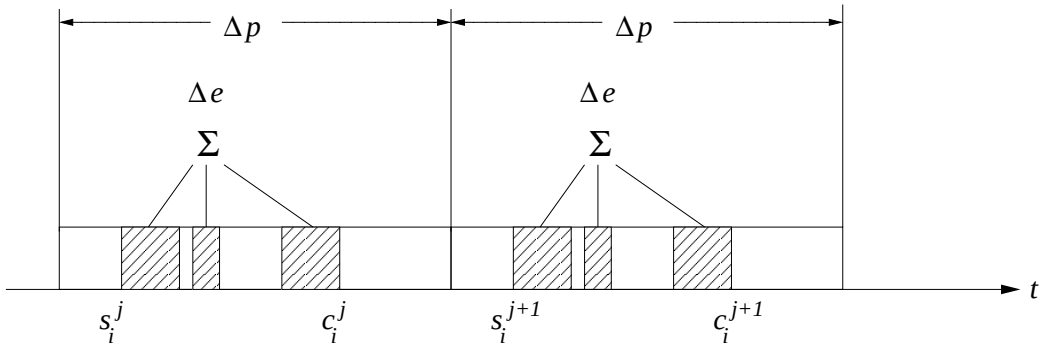

Abbildung 4.9: Die j-te und $(j+1)$-te unterbrochene Ausführung des Stellvertreterprozesses N_i^j und N_i^{j+1}.

weiter. Dennoch ist es Aufgabe des VM-Schedulers, die „faire" Verteilung der Rechenkapazitäten unter einer Menge gieriger Stellvertreterprozesse vorzunehmen. Dabei sollte die Verarbeitung in jedem der Gastsysteme für einen menschlichen Beobachter einigermaßen „flüssig" vonstatten gehen, und in jedem Fall sollte der eingesetzte Scheduler arbeitserhaltend sein, d.h. alle verfügbaren Rechenkapazitäten sind vollständig zu verteilen. Diese Anforderungen lassen sich mit Hilfe eines Verfahrens zur anteiligen Prozessorzuteilung (siehe 2.2.8) gut erfüllen. Tatsächlich verwenden VMware und Xen, wie auch die meisten anderen Virtualisierungsumgebungen ein solches Verfahren, da die Nicht-Echtzeitrechenlasten immer noch den klassischen Anwendungsfall virtueller Maschinen darstellen.

Bei der Ausführung von Nicht-Echtzeitbetriebssystemen auf einem Virtual Machine Monitor ergibt sich eine Besonderheit, die in der Modellbetrachtung in 3.1 noch nicht berücksichtigt wurde: In einem Gastsystem können jederzeit Situationen auftreten, bei denen –abgesehen vom „idle-"Prozess– keiner der vorhandenen Prozesse rechenwillig ist. Ohne Virtualisierung würde in einer solchen Situation der „idle-"Prozess einfach so lange ausgeführt, bis wieder ein anderer, „nützlicher" Prozess rechenwillig wird, d.h. der „idle-"Prozess würde die überschüssige Rechenzeit konsumieren. In einer Virtualisierungsumgebung mit mehreren Gastsystemen könnte diese Rechenzeit hingegen womöglich von einem anderen Gastsystem sinnvoller genutzt werden. Daher werden Nicht-Echtzeitbetriebssysteme in Virtualisierungsumgebungen in der Regel so modifiziert[1], dass sie beim Ausführen des „idle-"Prozesses den Prozessor freiwillig abgeben, dabei aber rechenwillig bleiben. Durch dieses *kooperative Scheduling* kann der VM-Scheduler den Prozessor anderen Nicht-Echtzeitgastsystemen reihum zuteilen. Da die Gast-

[1] Bei der Paravirtualisierung ist eine solche Modifikation ohne weiteres möglich. Bei der Vollvirtualisierung kann der gleiche Effekt u.U. durch eine entsprechende Simulation des im „idle-"Prozess in der Regel enthaltenen HALT-Befehls erzielt werden.

systeme nach Abgabe des Prozessors weiterhin rechenwillig bleiben, können sie nach wie vor als gierig angesehen werden.

Der „faire" VM-Scheduler kann mitsamt der unter ihm arbeitenden Nicht-Echtzeitbetriebssystem wiederum als Stellvertreterprozess aufgefasst werden. Da der Scheduler arbeitserhaltend ist, ist auch dieser Stellvertreterprozess gierig. Das angesprochene kooperative Scheduling findet nur innerhalb dieses Stellvertreterprozesses statt.

Fazit

Im diesem Abschnitt wurden Strategien zur Planung von Echtzeit- und Nicht-Echtzeitrechenlasten aufgezeigt. Dabei war das Ziel für die Echtzeitrechenlasten, eine fristgerechte Ausführung aller Echtzeitprozesse sicherzustellen. Da Echtzeitrechenlasten als periodische unterbrechbare Prozesse beschrieben werden können, ist dies durch Anwendung der klassischen Planungsverfahren für diesen Prozesstyp, beispielsweise EDF oder RMS, erreichbar. Für die Nicht-Echtzeitrechenlasten wurden die zu ihrer Ausführung verfügbaren Rechenkapazitäten ermittelt, wobei vorausgesetzt wurde, dass diese Rechenkapazitäten neben einem statisch eingeplanten Anteil im Wesentlichen die von den Echtzeitrechenlasten nicht verwendeten Überschüsse sind. Die Verteilung der Rechenkapazitäten für Nicht-Echtzeitrechenlasten kann durch einen hier nicht ausführlich spezifizierten „fairen" Scheduler erfolgen.

4.1.5 Koexistenz von Echtzeit- und Nicht-Echtzeitsystemen

Es bleibt nun zu klären, wie die dynamische Umwidmung von Rechenzeit zu bewerkstelligen ist. Eine einfache und weit verbreitete Methode hierzu, zu der in Abschnitt 4.1.1 mit LynxOS und RTAI bereits zwei Beispiele gezeigt wurden, ist die prioritätsbasierte Prozesssteuerung. Bei LynxOS wird für die Verteilung der Rechenzeit unter gleich priorisierten Nicht-Echtzeitanwendungen das Round-Robin-Verfahren verwendet, was einfach ist, aber auch bestimmte Nachteile mit sich bringt (siehe 4.1.1). Bei RTAI wurde diese Aufgabe dem besser dazu geeigneten Linux-Scheduler überlassen. Die resultierende Scheduler-Hierarchie (vgl. Abbildung 4.1) besteht aus einem prioritätsgesteuerten Scheduler auf unterster Ebene, der die Echtzeitprozesse direkt steuert, und dem Linux-Scheduler, der als „fairer" Scheduler alle Nicht-Echtzeitprozesse zu einem Stellvertreter zusammenfasst, der wiederum vom prioritätsgesteuerten Scheduler als „idle-"Prozess (d.h. als niedrigst-priorer Echtzeitprozess) behandelt wird. Diese Konstruktion erfüllt bereits zwei wichtige Anforderungen:

- Sämtliche von Echtzeitprozessen ungenutzten Rechenkapazitäten werden zur Ausführung von Nicht-Echtzeitprozessen umgewidmet.
- Die Verteilung der Rechenkapazitäten unter den Nicht-Echtzeitprozessen erfolgt „fair", d.h. jeder Prozess bekommt seinen proportionalen Anteil.

Der RTAI-Scheduler hat aber auch einen Nachteil, auf den in Abschnitt 4.1.1 bereits hingewiesen wurde: Es gibt keinen Mechanismus zur zeitlichen Beschränkung der Echtzeitprozesse. Da Echtzeitprozesse normalerweise nicht gierig sind, wurde ein solcher Mechanismus beim Design offenbar nicht für nötig erachtet. Dadurch sind fehlerhafte oder böswillige Echtzeitprozesse mit entsprechend hoher Priorität jederzeit in der Lage, den Prozessor zu monopolisieren. Die für die Virtualisierung wesentliche sichere Entkopplung zwischen den Gastsystemen wäre mit diesem Scheduler nicht zu bewerkstelligen. In [HPOS05] wird eine geeignete Lösung dieses Problems vorgestellt: Den Echtzeitanwendungen werden individuelle periodische Zeitfenster zugewiesen. Auf diese Weise wird die Planung forciert: Solange Echtzeitprozesse nicht mehr als die für sie eingeplanten Kapazitäten beanspruchen, bleiben die Zeitfenster ohne Wirkung. Wenn ein Prozess aber (böswillig oder unbeabsichtigt) versucht, mehr als die ihm zugesicherte Prozessorzeit zu nutzen, so wird er unterbrochen. Die Folgen dieser Übertretung bleiben somit auf den Prozess beschränkt, der sie begangen hat.

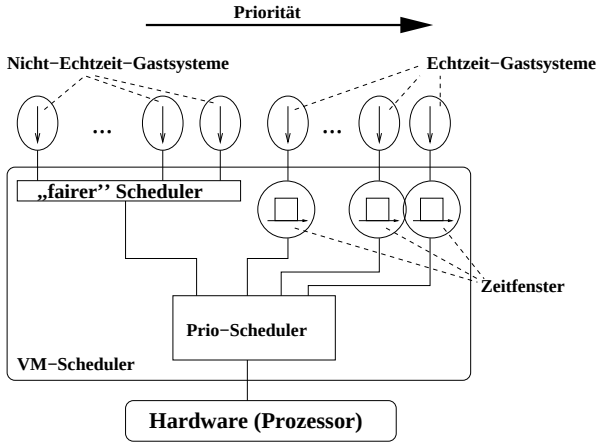

Abbildung 4.10: Prioritäts- und zeitgesteuerte Scheduler-Hierarchie.

Diese Scheduler-Architektur kann –mit kleinen Änderungen– in eine Virtualisierungsumgebung übertragen werden (siehe Abbildung 4.10): Die Echtzeitprozesse sind hier Echtzeit-Gastsysteme, die ihre Echtzeitprozesse als Stellvertreter kapseln. Die Nicht-Echtzeitprozesse werden zu Nicht-Echtzeit-Gastsystemen und das Linux-System, das bei RTAI die Umschaltung zwischen ihnen übernimmt, wird

hier auf einen „fairen" Scheduler reduziert, der die auf niedrigster Prioritätsstufe verfügbare Rechenkapazität unter ihnen aufteilt. Dieser Scheduler ist im Gegensatz zu RTAI nicht extern, sondern ein Teil des VM-Schedulers. Er kapselt alle Nicht-Echtzeit-Gastsysteme zu einem Stellvertreterprozess.

Den Echtzeit-Gastsystemen werden periodische Zeitfenster zugeordnet, deren Periodendauern Δp^i_{sv} und Ausführungszeiten Δe^i_{sv} nach den in 3.1.3 für RMS bzw. EDF gefundenen Regeln aus den Parametern der gekapselten Prozessmenge ermittelt werden. Solange keines der Gastsysteme mehr als die ihm zustehende Rechenzeit für sich beansprucht, bleibt diese Begrenzung ohne Wirkung und alle Systeme konkurrieren aufgrund ihrer Prioritäten gegeneinander. Mit Hilfe dieser Priorisierung wird die globale Planung durchgesetzt: Werden die Gastsysteme nach RMS geplant, so besitzt jedes von ihnen eine individuelle statische Priorität, die im umgekehrten Verhältnis zur Periodendauer festgelegt ist (vgl. Gleichung (4.10)). Bei globaler Planung nach EDF müssen die Prioritäten dynamisch so verändert werden, dass stets das rechenbereite Gastsystem mit dem kürzesten noch verbleibenden Zeitfenster die höchste Priorität erhält.

Mit dieser Scheduler-Konstruktion lässt sich nun auch die geforderte Fehlerlokalität umsetzen. Ihr fehlt jedoch noch eine wichtige Eigenschaft: Es besteht keine Möglichkeit, den Nicht-Echtzeitprozessen ein Mindestmaß an Rechenkapazität zuzusichern: Da sie auf niedrigster Echtzeit-Priorität angesiedelt sind, kommen sie nur dann zum Zug, wenn kein Echtzeitprozess Anspruch auf den Prozessor erhebt. Tritt dies nicht ein, so „verhungert" das Nicht-Echtzeitsystem. In 4.1.4 wurde festgestellt, dass auch für Nicht-Echtzeitsysteme eine gewisse Mindestauslastung U_{min} sichergestellt sein muss, damit ein minimaler Programmforschritt gewährleistet ist. Es gibt zwei Möglichkeiten, diese Mindestvergabe zu sichern:

1. *Nicht-Vergabe eines Teils der Rechenkapazität*: Die Zeitfenster der Echtzeitgastsysteme beschränken deren maximale Prozessorauslastung. Jedes Fenster ist beschrieben durch ein Wertepaar $(\Delta e^i_{sv}, \Delta p^i_{sv})$. Werden die Zeitfenster so gewählt, dass gilt:

$$1 - \sum_{i=1}^{n} \frac{\Delta e^i_{sv}}{\Delta p^i_{sv}} \geq U_{min} \tag{4.19}$$

   so wird ein Teil der verfügbaren Rechenzeit nicht an Echtzeitsysteme vergeben, sodass dieser Teil für die Nicht-Echtzeitverarbeitung übrig bleibt. Beim globalen Planen nach RMS können, je nach gefordertem U_{min}, bereits die ohnehin unvermeidlichen Planungslücken ausreichend sein, sodass keine besonderen Maßnahmen erforderlich sind.

Nachteilig hierbei ist allerdings, dass die Periodendauer, auf die sich die verfügbar gemachte Auslastung bezieht, dem kleinsten gemeinsamen Vielfachen der Periodendauern aller beteiligten Prozesse entspricht, das bekanntermaßen sehr hohe Werte annehmen kann.

2. *Prioritätsschub*: Ebenso wie allen Echtzeitgastsystemen kann auch dem Nicht-Echtzeitsystem ein periodisches Zeitfenster zugeordnet werden. Dieses Zeitfenster wird wie alle anderen in die Planung mit einbezogen, d.h. bei Planung nach RMS wird ihm eine statische Priorität entsprechend der Periodendauer zugewiesen, bei Planung nach EDF wird seine Priorität dynamisch erhöht, wenn es das kürzeste noch verbleibende Zeitfenster ist. Solange das Zeitfenster nicht erschöpft ist, erhält das Nicht-Echtzeitsystem einen „Prioritätsschub" sodass es mit der vom Echtzeit-Scheduler festgelegten Priorität des Zeitfensters arbeitet. Ist das Zeitfenster erschöpft, so nimmt der Nicht-Echtzeitprozess die minimale Priorität an und erhält somit bis zum Beginn seiner nächsten Periode nur noch die von den Echtzeitgastsystemen nicht genutzten Rechenkapazitäten.

Die zweite Methode entspricht der oben beschriebenen Vorgehensweise des „polling Server" und ist wegen der im Unterschied zur ersten Methode wählbaren Periodendauer in der Regel zu bevorzugen. Allerdings erfordert diese Methode zusätzliche Vorkehrungen seitens der Infrastruktur (das zusätzliche Zeitfenster mit der entsprechenden Sonderbehandlung).

4.1.6 Zusammenfassung

In Abschnitt 4.1 wurden die Aspekte Echtzeit und Virtualisierung gemeinsam betrachtet. Die Virtualisierung bedingt eine zweistufige Prozesshierarchie mit einem globalen VM-Scheduler und einem lokalen Scheduler pro Gastsystem. Das Zeitverhalten einer solchen Prozesshierarchie wurde im Hinblick auf die gleichzeitige Unterstützung von Echtzeit- und Nicht-Echtzeitanwendungen untersucht. Dazu wurden zunächst Beispiele von Betriebssystemen betrachtet, die *–ohne* Virtualisierungsschicht– Prozesshierarchien dazu nutzen, diese beiden Anwendungsklassen miteinander zu vereinen. Diese Betrachtungen lieferten sowohl Hinweise auf mögliche Lösungsansätze als auch Hinweise auf Probleme, die es zu lösen gilt. Anschließend wurden zwei existierende Virtualisierungsumgebungen – teils analytisch, teils experimentell– auf ihre Echtzeiteigenschaften hin untersucht. Dabei ergab sich, dass diese Systeme zumindest eingeschränkt tauglich für den Echtzeitbetrieb sind. Dieser durch Experimente gewonnene Eindruck konnte anhand einer einfachen Modellbetrachtung bestätigt werden. Als Nächstes wurden

Echtzeit- und Nicht-Echtzeitrechenlasten klassifiziert, und es wurden, basierend auf den in Kapitel 3 eingeführten Modellen, verschiedene Planungsverfahren für diese beiden Rechenlastklassen entwickelt. Zum Schluss wurde ausgehend von der Scheduler-Architektur des RTAI-Systems ein Konstrukt entwickelt, das die beiden Planungsverfahren zusammenführt. Dabei konnten mehrere inhärente Probleme des RTAI-Ansatzes identifiziert und beseitigt werden.

4.2 Echtzeitverarbeitung mit Mehrprozessorsystemen

In diesem Abschnitt werden die Aspekte der Echtzeitverarbeitung auf Mehrprozessorsystemen betrachtet. Anders als bei Einprozessorsystemen, wo die grundlegenden Konzepte seit langem vorliegen, ist dieses Thema Gegenstand aktueller Forschungen (z.B. [Fis07]). Dieser Abschnitt verfolgt jedoch nicht das Ziel, weitere neue Planungsverfahren für Mehrprozessorsysteme zu definieren und ggf. ihre Brauchbarkeit oder Optimalität formal zu beweisen. Vielmehr geht es hier darum, die Anforderungen, die solche Planungsverfahren an ihre unterlagerte Schicht stellen, zu identifizieren. Es sind diese Anforderungen, die von einem Virtual Machine Monitor zu erfüllen sind. Auch soll für die in aktuellen Forschungsarbeiten vorgeschlagenen Planungsverfahren die Brauchbarkeit für den praktischen Einsatz untersucht werden.

4.2.1 Klassifikation von Planungsverfahren

In [CFH+03] werden Mehrprozessor-Planungsprobleme und -algorithmen anhand zweier zueinander orthogonaler Kriterien klassifiziert. Eines dieser beiden Kriterien wurde bereits in den Abschnitten 2.2.7 und 2.2.8 eingeführt: Dort wurden Planungsverfahren anhand ihrer Einschränkungen hinsichtlich der dynamischen Prioritätenvergabe klassifiziert. An den Beispielen von RMS, EDF und anteiliger Zuteilung wurden die Unterschiede zwischen statischen, auftragsbezogen dynamischen und uneingeschränkt dynamischen Prioritäten gezeigt.

Freiheitsgrade bei der Prozessmigration

Eine ähnliche Unterscheidung wird in [CFH+03] hinsichtlich der Einschränkungen vorgenommen, denen ein Planungsverfahren bei der Migration von Prozessen unterliegt. Auch hier werden drei Klassen unterschieden:

1. *Keine Migration*: Es findet keine Prozessmigration statt: Prozesse bleiben während ihrer gesamten Lebensdauer an einen Prozessor gebunden (in [CFH+03] wird dieser Fall als *partitioned scheduling* bezeichnet).

2. *eingeschränkte Migration* (engl.: *restricted migration*): Jede einzelne Ausführung eines Prozesses findet auf genau einem Prozessor statt, verschiedene Ausführungen des Prozesses können jedoch auf verschiedenen Prozessoren ablaufen.

3. *uneingeschränkte Migration* (engl.: *unrestricted migration*): Ein Prozess kann jederzeit, also auch während einer Ausführung auf einen anderen Prozessor, verlagert werden.

Im ersten Fall (d.h. keine Migration) gibt es eine Warteschlange pro Prozessor. Jeder Prozess ist fest einer dieser Warteschlangen zugeordnet, d.h. alle seine Ausführungen werden in dieser prozessorbezogenen Warteschlange eingereiht. Im dritten Fall (uneingeschränkte Migration) gibt es eine globale Warteschlange. Die Ausführungen aller Prozesse des Systems werden in diese Warteschlange eingereiht, und alle Prozessoren arbeiten Aufträge aus dieser Warteschlange ab. Im zweiten Fall (eingeschränkte Migration) gibt es, wie im ersten Fall, eine Warteschlange pro Prozessor, allerdings wird zu den jeweiligen Bereitzeiten der Prozesse entschieden, in welche der Warteschlangen der aktuell anstehende Auftrag eingeordnet wird.

Klassifikation

Entsprechend ihrer Definition stellen auftragsbezogen dynamische Prioritäten eine Verallgemeinerung der statischen Prioritäten dar. Weiterhin sind uneingeschränkt dynamische Prioritäten eine Verallgemeinerung der eingeschränkt dynamischen Prioritäten. Die Unterscheidung zwischen eingeschränkt und uneingeschränkt dynamischen Prioritäten wird in der Literatur nur selten betont, da bereits das EDF-Verfahren, das auf eingeschränkt dynamischen Prioritäten basiert, nach [LL73] optimal ist. Im Falle von Mehrprozessorsystemen zeigt sich jedoch, dass Verfahren mit uneingeschränkt dynamischen Prioritäten den Verfahren mit eingeschränkt dynamischen Prioritäten überlegen sind ([CFH+03])

Die Einschränkungsgrade hinsichtlich der dynamischen Prioritätenvergabe und die Einschränkungsgrade hinsichtlich der Prozessmigration sind orthogonal, d.h. jeder kann mit jedem kombiniert werden. So ergeben sich $3 \times 3 = 9$ mögliche Klassen denen ein Planungsverfahren zugeordnet werden kann. In [CFH+03] werden Mehrprozessor-Planungsverfahren als „(x, y)-beschränkt" klassifiziert, wobei

Migration \Prioritäten	**1:** statisch	**2:** eingeschr. dyn.	**3:** uneingeschr. dyn.
3: uneingeschränkt	$(1,3)$-beschränkt	$(2,3)$-beschränkt	$(3,3)$-beschränkt
2: eingeschränkt	$(1,2)$-beschränkt	$(2,2)$-beschränkt	$(3,2)$-beschränkt
1: keine	$(1,1)$-beschränkt	$(2,1)$-beschränkt	$(3,1)$-beschränkt

Tabelle 4.1: Klassifikation von Mehrprozessor-Planungsverfahren (nach [CFH+03]).

$x, y \in \{1, 2, 3\}$. Eine Prozessmenge, die von einem (x, y)-beschränkten Planungsverfahren brauchbar eingeplant wird, wird im Folgenden als $F_{x,y}$ bezeichnet. Tabelle 4.1 zeigt die neun Klassen.

Beziehungen zwischen den Planungsverfahren

Carpenter et al stellen in [CFH+03] allgemein Beziehungen zwischen (x, y)-beschränkten Planungsverfahren her. Folgende Beziehungen werden definiert:

- *Überlegenheit*: Die Klasse der (w, x)-beschränkten Planungsverfahren ist der Klasse der (y, z)-beschränkten Planungsverfahren *überlegen*, wenn gilt: $F_{y,z} \subset F_{w,x}$, d.h. die Menge der unter dem (w, x)-beschränkten Verfahren planbaren Prozessmengen ist eine echte Teilmenge der unter dem (y, z)-beschränkten Verfahren planbaren Prozessmengen.
- *Äquivalenz*: Die Klasse der (w, x)-beschränkten Planungsverfahren ist der Klasse der (y, z)-beschränkten Planungsverfahren *äquivalent*, wenn gilt: $F_{y,z} = F_{w,x}$, d.h. die Menge der unter dem (w, x)-beschränkten Verfahren planbaren Prozessmengen ist auch unter dem (y, z)-beschränkten Verfahren planbar.
- *Unvergleichbarkeit*: Die Klasse der (w, x)-beschränkten Planungsverfahren ist der Klasse der (y, z)-beschränkten Planungsverfahren *nicht vergleichbar*, wenn gilt: $F_{y,z} \otimes F_{w,x}$, d.h. es existiert mindestens eine Prozessmenge, die unter dem (w, x)-beschränkten Verfahren, nicht aber unter dem (y, z)-beschränkten Verfahren planbar ist, und umgekehrt.

Tabelle 4.2 zeigt die von Carpenter et al ermittelten Beziehungen zwischen den nach (x, y)-beschränkten Verfahren planbaren Prozessmengen. (Für die formalen Beweise dieser Beziehungen sei auf [CFH+03] verwiesen.) Offensichtlich sind die $(3, 3)$-beschränkten Verfahren allen anderen überlegen: Jede Prozessmenge, die nach einem (x, y)-beschränkten Verfahren planbar ist, ist auch nach einem $(3, 3)$-beschränkten Verfahren planbar, d.h., es gilt:

	$F_{1,1}$	$F_{2,1}$	$F_{3,1}$	$F_{1,2}$	$F_{2,2}$	$F_{3,2}$	$F_{1,3}$	$F_{2,3}$	$F_{3,3}$
$F_{1,1}$	$=$	$\subset$	$\subset$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\subset$
$F_{2,1}$	$\supset$	$=$	$=$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\subset$
$F_{3,1}$	$\supset$	$=$	$=$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\subset$
$F_{1,2}$	$\otimes$	$\otimes$	$\otimes$	$=$	$\subset$	$\subset$	$\otimes$	unbek.	$\subset$
$F_{2,2}$	$\otimes$	$\otimes$	$\otimes$	$\supset$	$=$	$\subseteq$	$\otimes$	unbek.	$\subset$
$F_{3,2}$	$\otimes$	$\otimes$	$\otimes$	$\supset$	$\supseteq$	$=$	$\otimes$	$\otimes$	$\subset$
$F_{1,3}$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$=$	$\subset$	$\subset$
$F_{2,3}$	$\otimes$	$\otimes$	$\otimes$	unbek.	unbek.	$\otimes$	$\supset$	$=$	$\subset$
$F_{3,3}$	$\supset$	$\supset$	$\supset$	$\supset$	$\supset$	$\supset$	$\supset$	$\supset$	$=$

Tabelle 4.2: Beziehungen zwischen nach (x, y)-beschränkten Verfahren planbaren Prozessmengen (nach [CFH+03]).

$$F_{x,y} \subseteq F_{3,3}, \forall x, y \in \{1, 2, 3\} \tag{4.20}$$

Umgekehrt könnte man erwarten, dass die Menge $F_{1,1}$ der unter $(1, 1)$-beschränkten Verfahren planbaren Prozessmengen auch unter allen weniger stark eingeschränkten Planungsverfahren planbar sind, dass also $F_{1,1} \subseteq F_{x,y}, \forall x, y \in \{1, 2, 3\}$ gelte. Dies ist jedoch offensichtlich *nicht* der Fall. So sind beispielsweise alle Verfahren, die mit statischen Prioritäten arbeiten, und die unterschiedlichen Migrationsklassen angehören, nicht vergleichbar:

$$F_{1,x} \otimes F_{1,y}, \forall x, y \in \{1, 2, 3\}, x \neq y \tag{4.21}$$

Von statischen über eingeschränkt dynamische bis zu uneingeschränkt dynamischen Prioritäten bestehen zunehmend weniger Einschränkungen. Für einen gegebenen Migrationsgrad x gilt daher allgemein, dass:

$$F_{1,x} \subseteq F_{2,x} \subseteq F_{3,x}, \forall x \in \{1, 2, 3\} \tag{4.22}$$

Umgekehrt ergibt sich jedoch für verschiedene Migrationsgrade keine derartige Beziehung.

4.2.2 Eignung $(3, 3)$-beschränkter Planungsverfahren

Nach Gleichung (4.20) ist ein $(3, 3)$-beschänktes Planungsverfahren, also ein Verfahren mit uneingeschränkter Migration und uneingeschränkt dynamischen Prioritäten jedem anderen Verfahren überlegen. Das heißt, es ist in der Lage, für alle

Prozessmengen, die unter einem stärker eingeschränkten Verfahren planbar sind, ebenfalls einen Plan zu finden, und es findet darüber hinaus noch weitere Pläne, die von keinem der anderen Verfahren geliefert werden. Ein $(3, 3)$-beschänktes Planungsverfahren müsste also ein vorteilhafter Ansatz sein. Tatsächlich lässt sich zeigen (siehe [Z08], S. 242), dass ein Verfahren zur anteiligen Zuordung[1] für eine gegebene Prozessmenge P und ein Mehrprozessorsystem mit m Prozessoren genau dann einen brauchbaren Plan liefert, wenn gilt: $U(P) \leq m$. Bei diesen Überlegungen wurde allerdings von einigen vereinfachenden Annahmen ausgegangen, die in der Realität so nicht immer gegeben sind:

- *Unabhängigkeit der Prozesse*: In der Realität bestehen vielfach Abhängigkeiten zwischen verschiedenen Prozessen. Insbesondere wenn es sich um mittelgranulare Parallelprozesse handelt, gibt es zahlreiche Synchronisationspunkte, die zu lang andauernden Blockierungen führen können, falls die Teilprozesse eines Parallelprozesses nicht echt-parallel arbeiten. Hierauf nimmt das anteilige Zuteilungsverfahren keine Rücksicht.

- *Vernachlässigung der Umschaltverluste*: Für die Migration von Prozessen wurden keine Kosten veranschlagt. Tatsächlich ist aber der Vorgang der Migration eines Prozesses mit den gleichen Verlusten behaftet, wie die in 3.2.2 betrachteten Prozessumschaltungen: Bei einer Migration kann ein Prozess keine Cache-Inhalte zum neuen Prozessor „mitnehmen", d.h. der Cache auf dem neuen Prozessor muss erst geladen werden, was eine entsprechende Verlangsamung des Programmfortschritts zur Folge hat. Auch ist der Laufzeitaufwand durch den Scheduler nicht unerheblich, d.h. neben Cachebedingten Umschaltverlusten fallen auch beträchtliche Unterbrechungsverluste an.

Die Probleme mit einem Verfahren zur anteiligen Prozessorzuordnung sind in Mehrprozessorsystemen die gleichen wie im Einprozessorfall: Die Idealsituation einer kontinuierlichen, gleichmäßigen Lastverteilung kann nur durch ein infinitesimal kleines Quantum erreicht werden, was aufgrund der Umschaltverluste nicht realisierbar ist. Realistisch liegt eine sinnvolle Untergrenze des Quantums nach den Ergebnissen von Abschnitt 3.2.2 in der Größenordnung einiger Millisekunden. Echtzeitanwendungen deren Fristen in dieser Größenordnung oder darunter liegen, können nicht nach einem solchen Verfahren eingeplant werden. Wie bereits in Abschnitt 4.1.4 festgestellt, kann nur ein Teil des Anwendungsspektrums, das eine zur Konsolidierung eingebetteter Systeme geeignete Plattform unterstützen

[1] Die anteilige Zuordnung für Mehrprozessorsysteme gehört zu den $(3, 3)$-beschänkten Verfahren.

muss, nach der Methode der anteiligen Prozessorzuteilung geplant werden. Für Echtzeitanwendungen, deren Fristen in der Größenordnung des Quantums oder darunter liegen, müssen daher andere Planungsverfahren gewählt werden, die bezüglich des Migrationsgrades restriktiver sind.

4.2.3 $(2,1)$- und $(1,1)$-beschränkte Planungsverfahren

Systeme mit mehreren Prozessoren werden in aller Regel eingesetzt, um eine höhere Verarbeitungsleistung zu erzielen. Bei Nicht-Echtzeitsystemen wird diese Leistung gewöhnlich in Form eines Durchsatzes oder eines durchschnittlich pro Zeiteinheit erledigten Auftragsvolumens gemessen. Bei Echtzeitsystemen liegt das oberste Ziel in der Rechtzeitigkeit. Das $(3,3)$-beschränkte Planungsverfahren vereint –zumindest theoretisch– beide Zielsetzungen in idealer Weise miteinander: Es erreicht die maximal mögliche Auslastung und somit den maximalen Durchsatz, und zugleich garantiert es fristgerechte Ausführung. In der Praxis scheitert diese Vorstellung jedoch daran, dass das Quantum des Schedulers nicht beliebig klein gewählt werden kann. Andere Verfahren wie EDF oder RMS verwenden kein festes Quantum und haben dieses Problem daher nicht, d.h. sie können bei gleichermaßen leistungsfähiger Hardware wesentlich kürzere Fristen garantieren, ohne dabei exzessive Umschaltverluste in Kauf nehmen zu müssen. Allerdings kommt es bei diesen Verfahren zu Planungslücken, sodass die erreichbare Prozessorauslastung immer deutlich hinter dem Maximalwert zurückbleibt. In [CFH+03] werden als Vertreter der $(2,1)$- und $(1,1)$-beschränkten Planungsverfahren Varianten von EDF und RMS untersucht, die durch Hinzunahme eines „first-fit"- bzw. „best-fit"-Verfahrens zur Auswahl des Prozessors zu Mehrprozessor-Planungsverfahren ausgebaut wurden: Beim „first-fit" wird ein Prozess dem ersten Prozessor zugeteilt, der laut dem (Einprozessor-)Einplanbarkeitstest noch genügend Betriebsmittel hat, um den neuen Prozess aufzunehmen. Bei „best-fit" wird derjenige Prozessor ausgewählt, der (1) genügend freie Betriebsmittel zur Ausführung des Prozesses besitzt, und der (2) nach Annahme des neuen Auftrages die geringsten ungenutzt verbleibenden Betriebsmittel hat.

Für das ($(2,1)$-beschränkte) EDF-Verfahren mit „first-fit" oder „best-fit" wird in [CFH+03] gezeigt, dass eine Prozessmenge P auf einem System mit m Prozessoren planbar ist, wenn gilt:

$$U(P) \leq \frac{m+1}{2} \tag{4.23}$$

Für das ($(1,1)$-beschränkte) RMS-Verfahren mit „first-fit" gilt nach [OB98], dass eine Prozessmenge P auf einem System mit m Prozessoren planbar ist, wenn:

$$m \cdot \left(\sqrt{2} - 1\right) \leq U(P) \leq \frac{m+1}{1+2^{\frac{1}{m+1}}} \tag{4.24}$$

Beide Gleichungen geben untere Schranken der Auslastung an, d.h. unter günstigen Bedingungen kann die Auslastung auch wesentlich höher ausfallen. Dennoch muss bei Anwendung eines $(2,1)$- oder $(1,1)$-beschränkten Planungsverfahrens mit deutlichen Einbußen hinsichtlich der erzielbaren Auslastung gegenüber der anteiligen Zuteilung gerechnet werden. Nichtsdestotrotz kann die Anwendung dieser Verfahren im praktischen Einsatz vorteilhaft sein. Ihre Fähigkeit, kürzere Fristen ohne allzu große Verluste einhalten zu können, wurde bereits angesprochen. Darüber hinaus gibt es noch weitere Grunde, die für diese Verfahren sprechen:

- Da es keine Migrationen gibt, sind diese Verfahren –zumindest bei statischer Planung– grundsätzlich nicht auf SMP-Systeme beschränkt. Unter der Voraussetzung einer hinreichend präzise synchronisierten, gemeinsamen Zeitbasis (vgl. [Bom08]) kann auch in NUMA- oder NoRMA-Systemen nach diesen Verfahren geplant werden.

- Da hier jeder Prozessor einen eigenen Scheduler besitzt, besteht die Möglichkeit, zeit- und ereignisgesteuerte Prozesse voneinander zu entkoppeln. Das Problem des Konflikts zwischen Zeit- und Ereignissteuerung wurde bereits in Abschnitt 2.2.6 angesprochen: Sind von einem Echtzeitsystem sowohl zeit- als auch ereignisgesteuerte Aufgaben zu bewältigen, so muss in einem Einprozessorsystem eine dieser beiden Prozessklassen bevorzugt werden, was unmittelbar eine Verschlechterung der Echtzeiteigenschaften der nicht-bevorzugten Prozessklasse zur Folge hat. In einem Mehrprozessorsystem besteht nun die Möglichkeit, zeit- und ereignisgesteuerte Prozesse jeweils eigenen Prozessoren zuzuweisen, die dann den für „ihre" Klasse von Prozessen am besten geeigneten (d.h. zeit- oder ereignisgesteuerten) Scheduler verwenden. Der Gewinn besteht in hier erster Linie[1] in der Entkopplung: Da keine (oder weniger) Konflikte zwischen Echtzeitprozessen bestehen, können genauere Aussagen über deren Zeitverhalten gemacht werden, d.h. die Varianz von Start- und Abschlusszeiten wird verkleinert.

Daneben gilt, wie auch bereits für den Einprozessorfall in 4.1.5 betrachtet, dass die durch Planungslücken bedingte Verringerung der Auslastung nicht gleichbedeutend mit einem entsprechenden Verlust an Rechenleistung ist: Auch in Mehrpro-

[1] Daneben ist das System natürlich auch in der Lage, insgesamt mehr Aufträge pro Zeiteinheit abzuwickeln.

zessorsystemen können dynamisch anfallende, von keinem Echtzeitprozess genutzte Rechenkapazitäten dynamisch für Nicht-Echtzeitprozesse nutzbar gemacht werden.

4.2.4 Synchrone, Parallele Echtzeitverarbeitung

Einige Echtzeitaufgaben eignen sich besonders gut zur Parallelisierung. Ein gutes Beispiel hierfür ist die Verarbeitung von Bilddaten mit linearen Filteroperationen: Dabei kann jeder Teilprozess unabhängig einen Teil des Bildes bearbeiten. Im Idealfall[1] kann auf einem Rechensystem mit m physischen Prozessoren die Ausführungszeit jedes Teilprozesses und damit auch die Ausführungszeit des gesamten, parallelen Echtzeitprozesses auf $\frac{1}{m}$ reduziert werden. Für alle Teilprozesse gelten gemeinsame Bereitzeiten und Fristen, d.h. alle können zum gleichen Zeitpunkt gestartet werden, und der am längsten arbeitende Teilprozess bestimmt die Abschlusszeit. Abbildung 4.11 zeigt ein Beispiel eines Echtzeitprozesses, dessen Last in dieser Weise auf $m = 3$ Teilprozesse aufgeteilt wird. Unter der Voraussetzung, dass alle Teilprozesse parallel auf verschiedenen Prozessoren arbeiten, werden entsprechend kürzere Fristen (d_i' in Abbildung 4.11) erzielbar. Die kürzest-mögliche Frist ergibt sich für den Fall, dass die Teilprozesse exakt gleichzeitig gestartet werden und dass jeder von ihnen den gleichen Anteil an der zu verrichtenden Arbeit hat. Diese gezielte Parallel-Ausführung auf verschiedenen Prozessoren entspricht dem bereits eingeführten „Coscheduling".

Für das Abarbeiten eines solchen Prozesses ist keine Migration zur Laufzeit erforderlich: die Arbeit wurde vorab an die Anzahl zur Verfügung stehender Prozessoren angepasst und statisch zugeteilt. Ein $(1, 1)$- oder $(2, 1)$-beschränktes Planungsverfahren genügt für eine solche Planungsaufgabe vollkommen.

4.2.5 Zusammenfassung

In Abschnitt 4.2 wurden Verfahren zur Planung der Echtzeitverarbeitung auf Mehrprozessorsystemen anhand zweier orthogonaler Kriterien, dem Grad der dynamischen Prioritätenvergabe und dem Grad der Prozessmigration, klassifiziert. Die von Carpenter et al in [CFH+03] ermittelten Beziehungen zwischen den insgesamt 9 Klassen von Planungsverfahren wurden analysiert. Der sich daraus ergebende theoretische Idealfall eines sowohl hinsichtlich der Migration als auch der dynamischen Prioritätsvergabe unbeschränkten Planungsverfahrens wurde am

[1] Der allerdings in der Regel nicht in vollem Umfang erreicht wird – vgl. 2.1.3

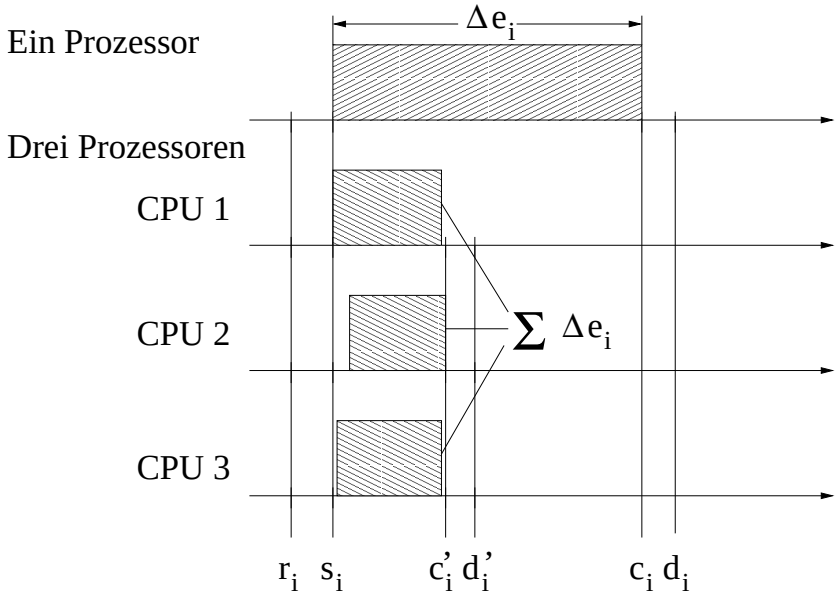

Abbildung 4.11: Beispiel: Parallelisierung eines Echtzeitprozesses.

Beispiel der anteiligen Prozessorzuteilung im Hinblick auf seine praktische Nutzbarkeit geprüft. Dabei ergab sich, wie schon in Abschnitt 4.1, dass dieses Verfahren nur für Nicht-Echtzeitaufgaben und für einen Teil der anstehenden Echtzeitaufgaben realistisch geeignet ist. Für Aufgabenstellungen, bei denen Fristen im Submillisekundenbereich einzuhalten sind, sind andere Planungsverfahren, die ohne Migration arbeiten, besser geeignet. Im Vergleich zur anteiligen Prozessorzuteilung erreichen diese zwar nur eine geringere Prozessorauslastung, aber dafür bieten sie eine bessere Zeittreue.

Wie schon in Abschnitt 4.2 für den Einprozessorfall festgestellt, zeigt sich auch hier die Notwendigkeit, die Anwendungen anhand ihrer Zeitanforderungen in zwei Klassen einzuteilen: Nicht-Echtzeitanwendungen sowie Echtzeitanwendungen, deren Zeitschranken groß gegen das Quantum eines Schedulers zur anteiligen Zuteilung sind, können unter einem solchen Scheduler arbeiten. Auf einem Mehrprozessorsystem unterliegt dieser Scheduler dabei keinen Beschränkungen bezüglich der Migration von Prozessen. Echtzeitanwendungen, deren Fristen in der Größenordnung des Quantums oder darunter liegen, arbeiten hingegen unter einem Scheduler mit statischen oder eingeschränkt dynamischen Prioritäten, der keine Prozessmigrationen zur Laufzeit unterstützt.

4.3 Virtualisierung von Mehrprozessorsystemen

In diesem Abschnitt werden die Aspekte der Virtualisierung und der Mehrprozessorsysteme gemeinsam betrachtet, wobei auch die Ergebnisse aus den Abschnitten 4.1 (Virtualisierung und Echtzeit) und 4.2 (Echtzeit und Mehrprozessorsysteme) mit einfließen.

4.3.1 Parallelität und Quasi-Parallelität

Auf einer Einprozessormaschine bildet ein Virtual Machine Monitor mehrere virtuelle Prozessoren (*vCPUs*) auf einen einzigen physischen Prozessor (*pCPU*) ab. Der Virtual Machine Monitor arbeitet dabei prinzipiell nicht anders als der Scheduler eines Betriebssystems: Die virtuellen Prozessoren, die er an seiner Schnittstelle zu den virtuellen Maschinen anbietet, sind Prozesse, d.h. Abstraktionen von Programmen in Ausführung. Diese virtuellen Prozessoren existieren dabei jeweils nur innerhalb sich nicht überschneidender Zeitfenster, da sie bei einem Einprozessorsystem reihum auf den einzigen physischen Prozessor abgebildet werden müssen.

Auch bei einem Mehrprozessorsystem werden virtuelle auf physische Prozessoren abgebildet, wobei hier jedoch mehrere physische Prozessoren zur Verfügung stehen. Dadurch können im Unterschied zu Einprozessorsystemen mehrere virtuelle Prozessoren gleichzeitig existieren: Gibt es m physische Prozessoren, so können von n virtuellen Prozessoren bis zu m Stück gleichzeitig arbeiten. Auf der Ebene der virtuellen Maschinen gibt es somit zwei Klassen von virtuellen Prozessoren: solche, die echt-parallel zueinander arbeiten, und solche, die zeitversetzt d.h. quasi-parallel arbeiten (Siehe Abbildung 4.12).

Viele Virtual Machine Monitore verbergen diesen Unterschied vor ihren Anwendern. Für ein Gastbetriebssystem ist es dann weder nachvollziehbar noch kontrollierbar, welche seiner zugewiesenen virtuellen Prozessoren parallel und welche quasi-parallel zueinander arbeiten. Der Virtual Machine Monitor kann sogar die Zuordnung von virtuellen zu physischen Prozessoren zur Laufzeit verändern, sodass virtuelle Prozessoren, die zu einem Zeitpunkt parallel arbeiten, zu einem anderen Zeitpunkt quasi-parallel arbeiten können

Wie in 3.3.4 bereits erläutert wurde, basiert die Konstruktion eines Mehrprozessorbetriebssystems auf der Annahme, dass die zur Verfügung stehenden Prozessoren parallel arbeiten. Diese Annahme ist beim Betrieb eines Mehrprozessorsystems als Gast einer Virtualisierungsumgebung nicht mehr erfüllt. Hieraus ergeben sich Probleme, die im Folgenden zu diskutieren sind.

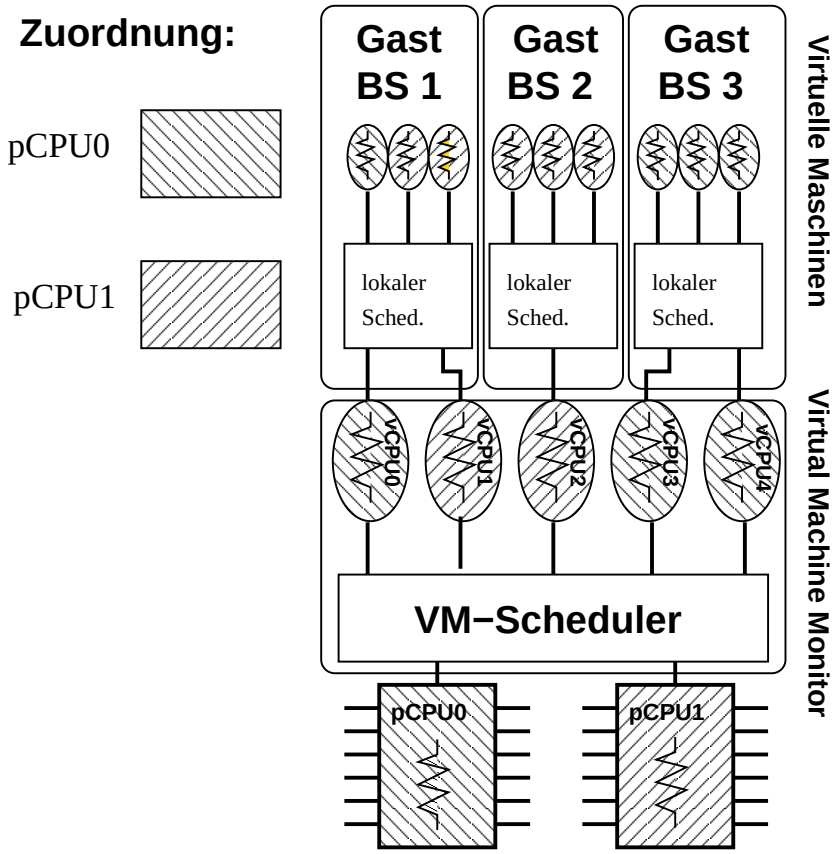

Abbildung 4.12: Echt-parallele und quasi-parallele virtuelle Maschinen.

4.3.2 Automatischer Lastausgleich

In [Z08] wird gezeigt, dass das $((3,3)$-beschränkte) Planungsverfahren der anteiligen Prozessorzuordnung für Mehrprozessorsysteme optimal ist. Dieses Verfahren hat (unter Anderem) uneingeschränkte Freiheiten bei der Migration von Prozessen (vgl. 4.2.1), d.h. es kann zu beliebigen[1] Zeitpunkten Prozessmigrationen vornehmen. Auf diese Weise wird eine anteilige Verteilung der Rechenlast über mehrere Prozessoren erreicht. Dies wird auch als Lastausgleich (engl.: *load balancing*) bezeichnet. Da dabei ein Prozess jederzeit und von ihm selbst unbemerkt auf einen anderen Prozessor verlagert werden kann, gibt es für ihn keine Möglichkeit zu entscheiden, mit welchen anderen Prozessen er um denselben Prozessor konkurriert, und welche Prozesse anderen Prozessoren zugeordnet sind. Aus Sicht der Pro-

[1] In der praktischen Umsetzung kann dieses Verfahren nicht zu wirklich beliebigen Zeitpunkten eingreifen, sondern es arbeitet mit einem endlichen Quantum, sodass mögliche Prozessverlagerungszeitpunkte immer Vielfache dieses Quantums sind.

zesse sind also keine Annahmen über Parallelität oder Quasi-Parallelität anderer Prozesse möglich.

Einige Virtual Machine Monitore (z.B. Xen) implementieren einen solchen Lastausgleich auf Ebene des VM-Schedulers mit dem Ziel, die Rechenlast automatisch (d.h. ohne dass dazu eine Intervention eines Systemadministrators nötig wäre) und gleichmäßig auf die verfügbaren physischen Prozessoren zu verteilen. Dies hat jedoch einige ggf. unerwünschte Folgen, je nachdem, ob es sich bei dem betrachteten Gastsystem um ein Ein- oder ein Mehrprozessorsystem handelt:

- Ein *Einprozessorbetriebssystem* kann als Gast einer Virtualisierungsumgebung stets nur einen einzigen virtuellen Prozessor nutzen. Dies gilt auch dann, wenn das System auf Anwendungsebene mehrere rechenbereite Prozesse besitzt. Wird ein Einprozessorsystem von einem Prozessor zu einem Anderen verschoben, so ändert sich dadurch nichts an der Gesamtauslastung des Systems: Die Rechenlast eines Einprozessorsystems kann immer nur als Ganzes bewegt werden, d.h. durch eine Verlagerung wird auf einem Prozessor immer genau so viel Kapazität frei, wie auf einem anderen neu belegt wird. Eine anteilige Verteilung der Last auf verschiedene Prozessoren kann daher nur erreicht werden, indem das Gastsystem jeweils für die Dauer entsprechend bemessener Zeitfenster auf dem einen oder anderen Prozessor verbleibt um anschließend auf einen anderen Prozessor verlagert zu werden. Daher muss der VM-Scheduler permanent Verlagerungen vornehmen. Um eine möglichst kontinuierliche Verteilung der Last zu erreichen, müssten die absoluten Verweildauern des Gastsystems auf den jeweiligen Prozessoren infinitesimal sein, was in der Realität aufgrund der mit Prozessorwechseln verbundenen Umschaltverluste nicht machbar ist. Die Situation ist analog zu der zeitlichen Verteilung von Rechenzeit (s. 2.2.8). Bezüglich sinnvoller Untergrenzen für Verweildauern gilt das unter 4.2.2 Gesagte: mit aktueller Hardware führen Zeitfenster in der Größenordnung einiger Millisekunden oder darunter zu oft nicht mehr vertretbaren Verlusten.

- Ein *Mehrprozessorsystem* besitzt in der Regel eine eigene Strategie zum Lastausgleich. Wie in 3.3 erläutert, werden Betriebssysteme für die Ausführung unter einem Virtual Machine Monitor –wenn überhaupt– nur sehr geringfügig verändert, sodass diese Strategie auch bei einem Mehrprozessorgastsystem vorhanden ist. Als Gast einer Virtualisierungsumgebung kann das Mehrprozessorsystem seine Strategie zum Lastausgleich nur auf die virtuellen Prozessoren anwenden, die ihm vom Virtual Machine Monitor zur Verfügung gestellt werden. Versucht nun der Virtual Machine Monitor seinerseits, die Lasten der virtuellen Prozessoren auf physische Prozessoren zu verteilen, so unterläuft er damit die Strategie des Gastsystems. Zweifellos

ist eine solche Überlagerung zweier lastausgleichender Scheduler, die zudem nicht aufeinander abgestimmt sind, problematisch. Je nachdem, nach welchen Kriterien die Scheduler vorgehen, kann es so zu zahlreichen unnötigen Prozessorwechseln kommen, die mit entsprechenden Kosten verbunden sind. Bestenfalls bleibt die Überlagerung ohne Wirkung, d.h. auf einer der beiden Ebenen kann der Lastausgleich ebensogut entfallen.

Der dynamische Lastausgleich auf Ebene des Virtual Machine Monitors ist also grundsätzlich als problematisch einzustufen: Bei einem Mehrprozessorsystem steht er im Konflikt mit dessen internem Lastausgleich. Da das Gastsystem im Zweifelsfall die Erfordernisse seiner Prozesse besser „kennt" als der Virtual Machine Monitor, sollte seine Strategie durchgesetzt werden, d.h. der Lastausgleich auf Seiten des VM-Schedulers sollte entfallen.

Bei Einprozessorsystemen ist ein Lastausgleich auf Ebene des VM-Schedulers erforderlich, um die Last des Systems gleichmäßig zu verteilen. Speziell bei Echtzeitsystemen bestehen aber mitunter andere, vorrangig zu behandelnde Anforderungen (zum Beispiel die in 4.2.3 genannte, zeit- und ereignisgesteuerte Prozesse jeweils getrennten Prozessoren zuzuordnen). Andererseits gibt es bei der statischen Planung von Echtzeitprozessen Situationen, in denen nur unter Inkaufnahme von Prozessmigrationen brauchbare Pläne zustande kommen. Da die Migrationen in diesem Fall statisch geplant sind, können die entstehenden Umschaltverluste in der Planung mitberücksichtigt werden. Eine dynamische Verlagerung wie zum Beispiel in Xen implementiert, sollte wegen der dann unkalkulierbaren Umschaltverluste für Echtzeitgastsysteme unterbleiben.

4.3.3 Anwendbarkeit des Liu-Layland-Modells

In der Arbeit von Popek und Goldberg ([PG74]) wurde definiert, dass eine virtuelle Maschine sich funktional genauso verhält wie die physische Maschine, die sie nachbildet, wobei jedoch das Zeitverhalten explizit ausgeklammert wurde. Eine Virtuelle Maschine *darf* also ein anderes Zeitverhalten zeigen, als ihr physisches Vorbild. Sieht man zunächst von Echtzeitanwendungen ab, so scheint die Gleichsetzung paralleler und quasi-paralleler virtueller Prozessoren, wie sie viele Virtual Machine Monitore betreiben, zunächst vollkommen legitim zu sein: Funktional ist jeder virtuelle Prozessor *für sich betrachtet* gleichwertig. Werden allerdings mehrere virtuelle Prozessoren gemeinsam betrachtet, so wird der Unterschied zwischen quasi-parallelen und echt-parallelen virtuellen Prozessoren –auch ohne Betrachtung von Echtzeitanwendungen– deutlich erkennbar. Solche Situation entstehen immer dann, wenn die Prozessoren gleichzeitig verschiedene Ausfüh-

rungspfade beschreiten, die zu einem gemeinsamen Programm gehören, d.h. wenn Parallelprozesse ausgeführt werden.

Durch die Parallelisierung von Programmen entsteht der Bedarf zur Kommunikation und Synchronisation der Teilprozesse. Diese müssen somit als voneinander abhängig angesehen werden. Sowohl die Modelle von Liu und Layland ([LL73]) als auch die in Abschnitt 4.2 vorgestellten Betrachtungen von Carpenter et al ([CFH+03]) sind zur Beschreibung solcher Problemstellungen nicht geeignet, da sie von der Grundvorstellung *unabhängiger* Prozesse ausgehen.

Insofern ist die Frage angebracht, welche praktische Relevanz die in den vorangegangen Kapiteln beschriebenen Modelle für die Ziele dieser Arbeit haben. Dazu ist zunächst festzustellen, dass die Gastsysteme, die in verschiedenen virtuellen Maschinen arbeiten, generell als voneinander unabhängig angesehen werden können (ihre sichere Entkopplung war ja gerade der Grund für die Einführung der Virtualisierungstechnik). Abhängigkeiten bestehen damit nur zwischen Prozessen, die gemeinsam in einer virtuellen Maschine arbeiten, also den Prozessen, die den lokalen Schedulern der einzelnen Gastsysteme unterliegen. Handelt es sich bei einem Gastsystem um ein Einprozessorsystem, so werden diese Abhängigkeiten bereits durch dessen lokalen Scheduler behandelt. Aus Sicht des Virtual Machine Monitors stellt ein Einprozessorgastsystem einen einzelnen Prozess dar, der, da das Gastsystem in einer eigenen virtuellen Maschine arbeitet, von anderen Prozessen unabhängig sein muss. Somit können nur in Mehrprozessorgastsystemen Abhängigkeiten auftreten, die möglicherweise die Anwendbarkeit des Liu-Layland-Modells gefährden. Ob dies für eine gegebene Prozessmenge, der Fall ist, muss jeweils geprüft werden. Eine solche Menge voneinander abhängiger Prozesse ist in jedem Fall nicht einfach durch Ausführungszeiten und Periodendauern beschreibbar. Ein VM-Scheduler für eingebettete Systeme muss auch für solche Rechenlasten geeignete Methoden anbieten, die es einerseits gestatten, diesen Rechenlasten hinreichende Betriebsmittel zur fehlerfreien Ausführung zuzuteilen, und die andererseits andere, mit den bekannten Modellen beschreibbare Prozessmengen nicht in ihrer Ausführung behindern. In 4.3.5 werden hierzu einige Vorgehensweisen angegeben. Zunächst werden jedoch unterbrechbare periodische Prozesse unter Anwendung des Liu-Layland-Modells betrachtet.

4.3.4 Planung unterbrechbarer periodischer Prozesse

In 4.1.4 wurde eine Unterscheidung von Echtzeit- und Nicht-Echtzeitrechenlasten getroffen. Betriebssysteme wurden in 3.3 als Ein- oder Mehrprozessorsysteme klassifiziert. Diese beiden Unterscheidungskriterien sind orthogonal zueinander, sodass hier insgesamt $2 \times 2 = 4$ Klassen von Gastbetriebssystemen zu betrachten

sind. Für diese vier Klassen werden im Folgenden die Anforderungen zusammengestellt, die sie jeweils an einen unterlagerten VM-Scheduler haben:

Sei also $P = \{1, ..., n\}$ eine Menge unterbrechbarer periodischer Prozesse mit den Periodendauern Δp_i, den Ausführungszeiten Δe_i, $i = 1...n$ und der sich daraus nach Gleichung (2.3) ergebenen Auslastung $U(P)$.

1. *Einprozessor-Echtzeitgastsysteme*

   Wird die Prozessmenge P unter einem Einprozessor-Gastbetriebssystem nach RMS eingeplant, so kann sie nach 3.1.3 als Ganzes gebenüber einem Virtual Machine Monitor durch einen Stellvertreterprozess mit der Periodendauer Δp_{sv} nach Gleichung (3.13) und der Ausführungszeit Δe_{sv} nach Gleichung (3.15) substituiert werden. Wird sie nach EDF eingeplant, so lässt sich analog nach den in 3.1.3 angegeben Methoden ein Wertepaar $(\Delta e_{sv}, \Delta p_{sv})$ ermitteln, das einen äquivalenten unterbrechbaren Stellvertreterprozess beschreibt.

   Dieser Stellvertreterprozess ist nun ein Prozess in der Menge V unterbrechbarer periodischer Prozesse, die der VM-Scheduler seinerseits einplant. Bei einem Einprozessorsystem kann diese Planung wiederum nach dem RMS- oder dem EDF-Verfahren erfolgen. Für ein Mehrprozessorsystem erfolgt die Planung nach einem $(2, 1)$- oder $(1, 1)$-beschränkten Verfahren [1], z.B. EDF mit „first-fit", EDF mit „best-fit" oder RMS mit „first fit". Die Planbarkeit kann dabei, je nach gewähltem Verfahren anhand einer der Gleichungen (2.6), (2.8), (4.24) oder (4.23) geprüft werden.

2. *Mehrprozessor-Echtzeitgastsysteme*

   Wird die Prozessmenge P unter einem Mehrprozessorsystem mit m Prozessoren wie in 4.2.3 beschrieben nach einem $(2, 1)$- oder $(1, 1)$-beschränkten Verfahren geplant, so nimmt dieses Verfahren eine statische Verteilung der Prozesse auf die zur Verfügung stehende Anzahl an Prozessoren vor. Da dieses Verfahren keine Migrationen zulässt, wird jeder Prozess $i \in P$ fest einem Prozessor zugeteilt. Dadurch wird die Prozessmenge in m Teilmengen P_k, $k = 1...m$ aufgespalten, die jeweils auf dem k-ten Prozessor ausgeführt werden. Diese Teilmengen P_k sind wiederum Mengen unterbrechbarer periodischer Prozesse. Für jede von ihnen kann somit nach den in 3.1.3 beschriebenen Methoden ein äquivalenter unterbrechbarer Stellvertreterprozess angegeben werden.

[1] Grundsätzlich könnte auch nach einem $(2, 2)$- oder $(1, 2)$-beschränkten Verfahren geplant werden. Carpenter et al geben in [CFH+03] auch für diesen Fall Planbarkeitskriterien an.

Diese m Stellvertreterprozesse sind nun in der Menge V unterbrechbarer periodischer Prozesse enthalten, die der VM-Scheduler seinerseits einplant. Die Planung erfolgt nach einem $(2, 1)$- oder $(1, 1)$-beschränkten Verfahren, z.B. EDF mit „first-fit", EDF mit „best-fit" oder RMS mit „first fit". Die Planbarkeit kann dabei anhand der Gleichungen (4.24) oder (4.23) entschieden werden.

3. *Einprozessor-Nicht-Echtzeitgastsysteme*

   Sämtliche Nicht-Echtzeitgastsysteme werden, wie in Abschnitt 4.1.5 beschrieben, durch einen „fairen" Scheduler gekapselt, der Teil des VM-Schedulers ist (siehe Abbildung 4.10 in 4.1.5). Ein Einprozessor-Gastsystem stellt gegenüber diesem Scheduler einen einzigen, gierigen Prozess dar. Dieser wird nach einem $(3, 3)$-beschränkten Verfahren geplant, d.h. der VM-Scheduler teilt dem Gastsystem einen proportionalen Anteil an den insgesamt für Nicht-Echtzeitsysteme verfügbaren Rechenkapazitäten zu und migriert das Gastsystem je nach Bedarf zu beliebigen physischen Prozessoren.

4. *Mehrprozessor-Nicht-Echtzeitgastsysteme*

   Auch Mehrprozessor-Nicht-Echtzeitsysteme werden durch den „fairen" Scheduler gekapselt. Ein Mehrprozessor-Gastsystem mit m Prozessoren stellt gegenüber diesem Scheduler einen Menge von m gierigen Prozessen dar. Diese wird nach einem $(3, 1)$-beschränkten Verfahren eingeplant, d.h. der VM-Scheduler teilt dem Gastsystem einen proportionalen Anteil an den insgesamt für Nicht-Echtzeitsysteme verfügbaren Rechenkapazitäten zu, wobei er aber *keine* Migrationen vornimmt (zur Begründung siehe 4.3.2).

Die vom „fairen" Scheduler an die Gesamtheit aller Nicht-Echtzeitsysteme verteilte Rechenkapazität wird aus drei verschiedenen Quellen gespeist (vgl. 4.1.4). Bei einem Mehrprozessorsystem fallen alle drei Quellen (zugewiesenes Zeitfenster, „Verschnitt" aus Planungslücken und nicht genutzte Ausführungszeiten) prozessorbezogen an. Bei einem Einprozessorsystem verlagert der VM-Scheduler den zugehörigen Stellvertreterprozess bei Bedarf auf den Prozessor, auf dem die freie Rechenkapazität verfügbar ist. Bei einem Mehrprozessorsystem wird vom lokalen Scheduler erwartet, dass er wenn nötig eine Migration vornimmt.

4.3.5 Zuteilung für abhängige Prozesse

In Abschnitt 4.3.4 wurden Vorgehensweisen für die Planung unabhängiger Prozesse in Virtualisierungsumgebungen angegeben. Dieses Prozessmodell deckt

einen Großteil der Anwendungen ab, die auf ein eingebettetes System zukommen. Speziell bei Mehrprozessorsystemen entstehen durch die Parallelisierung von Programmen aber auch voneinander abhängige Prozesse, für die die bisher vorgestellten Planungsmethoden nicht gut geeignet sind.

Das Problem der „Lock Holder Preemption"

Im Abschnitt 3.3.1 wurde gezeigt, dass die Synchronisation von Teilprozessen in einem Mehrprozessorbetriebssystem auf unterschiedliche Weise erfolgen muss, je nachdem, ob die zu synchronisierenden Prozesse parallel auf verschiedenen Prozessoren oder zeitversetzt auf demselben Prozessor arbeiten. Die zur Unterscheidung notwendige Zuordnung virtueller Prozessoren zu physischen Prozessoren unterliegt jedoch bei vielen Virtual Machine Monitoren der Kontrolle des VM-Schedulers, sodass ein Gastsystem die Zuordnung in der Regel nicht beeinflussen kann.

Ein Xen-basiertes System erlaubt sogar Konfigurationen, bei denen einem einzigen Gastsystem mehr virtuelle Prozessoren zugewiesen werden, als physische Prozessoren existieren. Im Xen-Handbuch wird zwar ausdrücklich vor solchen Konfigurationen gewarnt, jedoch fehlt dort eine Erklärung, welche Probleme im Einzelnen zu erwarten sind (vgl. [Xen05], S. 39).

Diese Erklärung findet sich in [ULSD04]: Hier wird eine Virtualisierungsumgebung für Mehrprozessorsysteme beschrieben, die nicht auf Xen, sondern auf dem Mikrokern „L4" [Lie95] basiert. Der Mikrokern übernimmt in diesem System die Rolle des Virtual Machine Monitors und stellt eine Anzahl von Prozessen als virtuelle Prozessoren zur Verfügung.

In [ULSD04] wird ausschließlich Linux, ein Mehrprozessorbetriebssystem, als Gastsystem betrachtet. Dabei zeigt sich ein als „lock holder preemption" bekanntes Problem: Da einer virtuellen Maschine jederzeit ein physischer Prozessor entzogen werden kann, kann dies auch dann geschehen, wenn die betroffene virtuelle Maschine gerade ein Spinlock hält. Wenn nun ein anderer Pfad (d.h. ein anderer virtueller Prozessor) derselben virtuellen Maschine versucht, dasselbe Spinlock zu belegen, so wartet er *aktiv* auf dessen Freigabe. Das aktive Warten geschieht hier unter der Annahme, dass ein Spinlock immer nur für sehr kurze Zeiten (wenige Mikrosekunden) gehalten wird. Ohne Virtualisierung trifft dies auch zu, da die mit Spinlocks geschützten kritischen Sektionen kurz und nicht-unterbrechbar sind. Werden aber stattdessen virtuelle Prozessoren verwendet, so wird diese Annahme verletzt. Das aktive Warten, das bei physischen Prozessoren nur wenige Maschinenzyklen verbraucht, dauert dann bis zu einigen zehn Millisekunden. In der Folge können unter Umständen erhebliche Mengen an Rechenkapazität durch aktives

Warten verschwendet werden. Wie schwer diese Folgen im Einzelnen sind, hängt von der Häufigkeit ab, mit der Synchronisationen zwischen den Teilprozessen des Gastsystems vorgenommen werden. In [Fri08] werden hierzu einige Messergebnisse vorgestellt, die am Beispiel eines auf Xen basierten Linux-Betriebssystems ermittelt wurden. Der dort anhand verschiedener Benchmarks festgestellte Verlust an Rechenleistung liegt in der Größenordung von 8%.

Die Ursache des Problems liegt in der Verletzung der dem Entwurf des Gastsystems zugrunde liegende Annahme, dass die virtuellen Prozessoren zu jeder Zeit parallel arbeiten. Ausgehend von dieser Annahme verwendet das Linux-System den nur für parallele Prozesse geeigneten Spinlock-Mechanismus zur Synchronisation. Damit gibt es zwei mögliche Lösungswege:

1. Anpassen der Laufzeitumgebung (d.h. hier: des VM-Schedulers) so, dass Parallelität der virtuellen Prozessoren immer gegeben ist.
2. Anpassung des Gastsystems dahingehend, dass die Entwurfsvoraussetzung paralleler Prozessoren fallengelassen werden kann.

Der erste Lösungsweg läuft auf ein Coscheduling auf der Ebene des VM-Schedulers hinaus, d.h. alle virtuellen Prozessoren einer virtuellen Maschine werden stets zeitgleich aktiviert. Der zweite Lösungsweg bedingt eine Modifikation des Gastsystems. Diese beiden Lösungswege werden im Folgenden diskutiert.

Lösung durch Coscheduling

Beim Coscheduling ([Ous80]) werden alle Teilprozesse eines Parallelprozesses als Einheit betrachtet und gleichzeitig auf verschiedenen Prozessoren zur Ausführung gebracht, d.h. ihre Startzeiten werden miteinander synchronisiert. Das beschriebene Problem der „lock holder preemption" kann vermieden werden, indem der VM-Scheduler alle virtuellen Prozessoren einer virtuellen Maschine Coscheduling anwendet. Auf diese Weise wird erzwungen, dass die virtuellen Prozessoren eines Gastsystems stets parallel arbeiten, sodass die Annahmen des Betriebssystems wieder Gültigkeit haben.

Ein anderer Anwendungsfall für Coscheduling wäre ein Gastbetriebssystem. das an seiner Benutzerschnittstelle spezielle Mechanismen für Parallelprozesse bietet. Ein Beispiel dazu ist das „CM*"-System, auf das sich die Arbeit von Ousterhout ([Ous80]) bezieht. Bei diesem System können sich Prozesse gegenüber dem Betriebssystem als zu einer „Task Force" gehörige Teilprozesse ausweisen. Das Betriebssystem räumt diesen Teilprozessmengen daraufhin eine Sonderstellung ein,

indem es sie an jeweils eigene physische Prozessoren bindet und stets gleichzeitig startet. Um eine solche Schnittstelle als Gast innerhalb einer virtuellen Maschine anbieten zu können, muss das Betriebssystem die Gewissheit haben, dass alle ihm zur Verfügung stehenden virtuellen Prozessoren parallel arbeiten, d.h. der VM-Scheduler muss seinerseits Coscheduling auf die virtuellen Prozessoren des Gastsystems anwenden.

Die meisten heutigen Mehrprozessor-Betriebssysteme unterstützen keinen derartigen Mechanismus, d.h. alle Anwenderprozesse werden als gleichwertige, unabhängige sequenzielle Prozesse angesehen. Selbst Programmierumgebungen wie OpenMP ([CJP07]), die das Erzeugen paralleler Programme zum Ziel haben, bilden derzeit aus Kompatibilitätsgründen ihre Teilprozesse auf Threads ab und verwenden z.B. zur Implementierung wechselseitiger Ausschlüsse verhältnismäßig aufwändige, potenziell blockierende Betriebssystemfunktionen. Nach [Ous80] wäre für solche mittelgranulare Parallelverarbeitung eine Synchronisation durch aktives Warten – unter der Voraussetzung, dass alle beteiligten Teilprozesse parallel arbeiten – effizienter.

Obwohl also die meisten Betriebssysteme heute an ihrer Anwendungsschnittstelle keine Funktionalität zur Kontrolle der Prozessorbindung bieten[1], setzen sie –wie am Beispiel der „lock holder preemption" gesehen– dennoch in ihrer Implementierung paralleles Arbeiten ihrer Prozessoren voraus. Darüber hinaus ist zu erwarten, dass im Zuge der weiteren Verbreitung von Multicore-Prozessoren und im Zuge ansteigender Zahlen von Prozessorkernen pro Prozessorchip die Forderung nach Schnittstellen zum effizienten Betrieb paralleler Prozessen auf Betriebssysteme zukommen wird.

Aus diesen Gründen sollte ein VM-Scheduler die Möglichkeit bieten, Coscheduling auf die virtuellen Prozessoren einer virtuellen Maschine anwenden zu können[2]. Die technischen Möglichkeiten hierzu sind leicht bereitzustellen: In 4.1.5 wurde bereits ein –dort noch auf Einprozessorsysteme beschränktes– Funktionsmodell des VM-Schedulers gezeigt (siehe Abbildung 4.10). In diesem Modell sind bereits Zeitfenster eingeführt, ursprünglich zu dem Zweck, die den Echtzeit-Gastsystemen zugewiesenen Zeitfenster zu überwachen. Bei der Erweiterung dieses Modells auf Mehrprozessorsysteme erhält jeder virtuelle Prozessor ein eigenes Zeitfenster. Indem (1.) sichergestellt wird, dass jeder virtuelle Prozessor einem eigenen physischen Prozessor zugewiesen ist (d.h. alle virtuellen Prozessoren arbeiten parallel) und indem (2.) die Zeitfenster der virtuellen Prozessoren synchro-

[1] Es gibt allerdings neben „CM*" noch einige weitere, im Bereich der Forschung eingesetzte Betriebssysteme, die solche Funktionen bieten (z.B. „processor sets" in Mach).

[2] Dieser Trend wird von neueren Versionen des Virtual Machine Monitors VMware bereits aufgegriffen: Nach [VMw05a] implementiert die aktuelle Version von VMware ESX Server Coscheduling.

nisiert werden, wird Coscheduling ermöglicht.

Lösung durch Anpassung des Gastsystems

In [ULSD04] wird Coscheduling zur Lösung des „lock holder preemption" Problems explizit ausgeschlossen, da, wie es dort heißt, Coscheduling virtuelle Prozessoren aktiviere, gleichgültig, ob für diese eine Aufgabe vorliegt oder nicht. Dies führe zu einer unzureichenden Auslastung einzelner Prozessoren. Tatsächlich sind Situationen möglich, in denen ein Mehrprozessorbetriebssystem aktuell weniger rechenwillige Prozesse als virtuelle Prozessoren hat. In solchen Situationen schreibt das Prinzip der Arbeitserhaltung vor, dass die nicht beschäftigten virtuellen Prozessoren ihre zugeordneten physischen Prozessoren abgeben, damit sie ggf. von anderen virtuellen Maschinen genutzt werden können. Coscheduling würde hingegen erzwingen, dass alle virtuellen Prozessoren einer virtuellen Maschine gleichzeitig ihre physischen Prozessoren abgeben, d.h. die nicht beschäftigten unter ihnen müssten in Ermangelung sinnvoller Aufgaben ihre Rechenkapazitäten durch Ausführen des „idle-"Prozesses verschwenden, bis die Zeitscheibe der virtuellen Maschine abgelaufen ist. Somit führt sowohl die „lock holder preemption" selbst als auch ihre Vermeidung mit Hilfe von Coscheduling zu Verschwendung von Rechenkapazität. Man beachte, dass diese Problematik nur Nicht-Echtzeitgastsysteme betrifft: Ein Echtzeitgastsystem ist nicht gierig und gibt den Prozessor ab, sobald es keinen Rechenzeitbedarf hat (es enthält keinen „idle-"Prozess).

Das in [ULSD04] betrachtete Linux-Gastsystem unterstützt auf Anwenderebene keine parallelen Prozesse: es gibt nur grobgranulare Parallelverarbeitung, d.h. weitgehend eigenständige Prozesse mit allenfalls „weichen" Echtzeitanforderungen, die nur selten Synchronisationsbedarf haben. Für Linux-Anwendungen ist es daher tatsächlich unerheblich, ob sie echt-parallel oder zeitversetzt ausgeführt werden. Nur wenn Synchronisationen zwischen Prozessen notwendig werden (d.h. hier: auf der Ebene des Linux-Betriebssystemkerns), muss zwischen diesen beiden Fällen unterschieden werden. Die Arbeit [ULSD04] stellt zwei verschiedene Möglichkeiten zur Lösung des Problems vor:

1. *Helfende Spinlocks*: Hierbei wird der Spinlock-Mechanismus erweitert: Sobald ein Prozess erkennt, dass der Halter eines Spinlocks, auf den er wartet, unterbrochen wurde, blockiert er selbst und spendet seine Rechenzeit (bzw. seinen physischen Prozessor) an den Halter des Spinlock, damit dieser weiterarbeiten und das Spinlock umgehend freigeben kann.

2. *Verhindern von lock holder preemption*: Hierzu wird der VM-Scheduler informiert, wenn ein Prozess im Begriff ist, einen Spinlock zu belegen. Der

VM-Scheduler vermeidet es daraufhin, den betreffenden Prozess in einem anschließenden Zeitintervall passend gewählter Dauer zu unterbrechen.

Beide Methoden beinhalten eine Anpassung des Gastsystems an die geänderten Bedingungen, die bei seiner Ausführung in einer virtuellen Maschine herrschen. Hierzu muss das Gastsystem die Kontrolle über die Zuweisung physischer Prozessoren beeinflussen können, und es muss mit dem VM-Scheduler interagieren. Im ersten Fall muss der Gast erkennen können, wer derzeit der Halter des Spinlock ist, ob dieser Halter unterbrochen wurde und wie lange die Unterbrechung noch andauern wird. Außerdem muss der Gast den VM-Scheduler dazu bewegen können, diesen Halter an seiner Statt zu aktivieren. Im zweiten Fall muss der Eigentümer des Spinlock dem VM-Scheduler anzeigen, dass Unterbrechungen unerwünscht sind.

4.4 Zusammenfassung: Anforderungen an einen VM-Scheduler

Zum Abschluss dieses Kapitels wird hier das Funktionsprinzip eines VM-Schedulers skizziert, der alle in Kapitel 4 herausgearbeiteten Anforderungen erfüllen kann.

Abbildung 4.13 zeigt das Funktionsprinzip des VM-Schedulers. Es handelt sich dabei um eine erweiterte Form des bereits in 4.1.5 für Einprozessorsysteme skizzierten Schedulers. Auch hier enthält der VM-Scheduler intern zwei Scheduler, von denen einer für die Echtzeitverarbeitung zuständig ist, während ein anderer die gleichmäßige Verteilung der den Nicht-Echtzeitprozessen zugeteilten Rechenzeit regelt. Es folgen nun stichwortartige Erklärungen zur Arbeitsweise dieses Schedulers.

- Ein $(3, 3)$- bzw. $(3, 1)$-beschränkter Scheduler fasst die virtuellen Prozessoren aller Nicht-Echtzeitgastsysteme zu einem einzigen, gierigen Stellvertreterprozess zusammen. Dabei wird für die Mehrprozessorgastsysteme *keine* Prozessmigration vorgenommen, da diese Gastsysteme intern eigene Strategien zur Zuordnung ihrer Prozesse zu Prozessoren besitzen. Für diese Klasse von Gastsystemen ist der Scheduler also $(3, 1)$-beschränkt.

- Einprozessorgastsysteme verwenden nur einen virtuellen Prozessor. Daher sorgt der –für diese Prozessklasse– $(3, 3)$-beschränkte Scheduler durch entsprechende Migrationen für eine gleichmäßige Verteilung dieser Rechenlast auf die zur Verfügung stehenden virtuellen Prozessoren

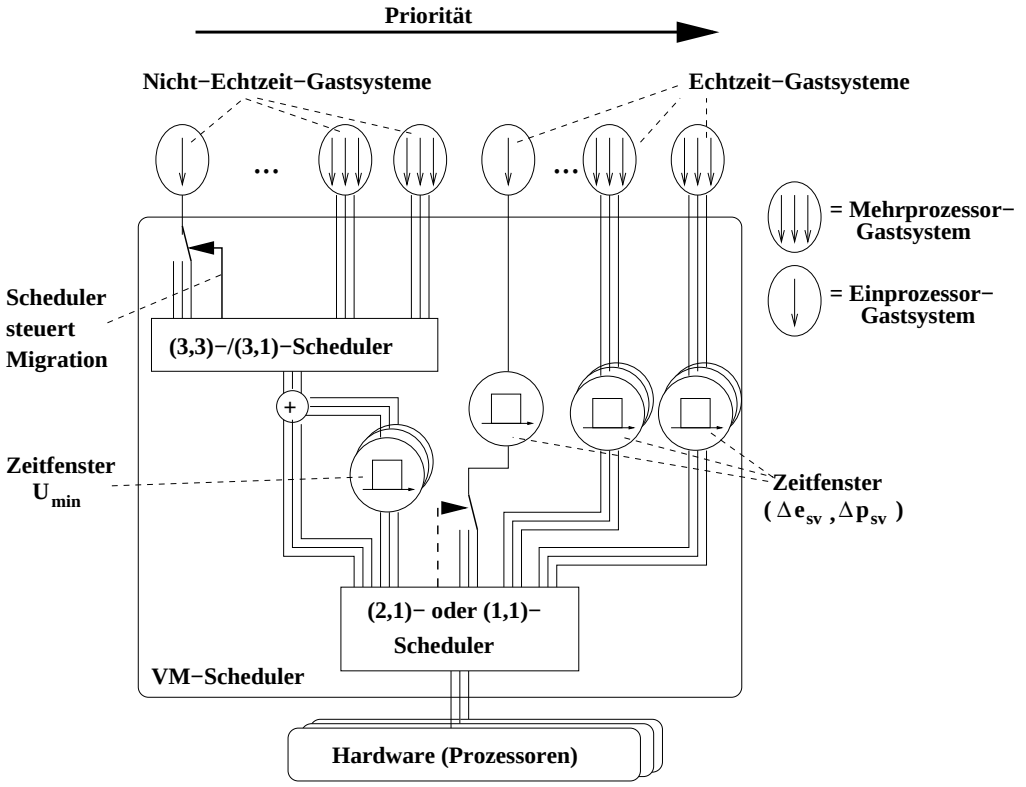

Abbildung 4.13: Prioritäts- und zeitgesteuerte Scheduler-Hierarchie für Mehrprozessorrechner.

- Ein $(2, 1)$- oder $(1, 1)$-beschränkter Scheduler bildet die virtuellen Prozessoren der Echtzeit-Gastsysteme und die des Stellvertreterprozesses der Nicht-Echtzeitgastsysteme auf die physischen Prozessoren der Hardware ab. Als Algorithmen für diesen Scheduler kommen z.B. die $(2, 1)$-beschränkten Verfahren EDF mit „first-fit" oder EDF mit „best-fit", oder das $(1, 1)$-beschränkte RMS mit „first-fit" in Frage. Das verwendete Verfahren entscheidet über das anzuwendende Planbarkeitskriterium.

- Für die Mehrprozessor-Echtzeitgastsysteme nimmt dieser Scheduler keine Migrationen vor, d.h. die virtuellen Prozessoren der Gastsysteme arbeiten parallel.

- Der virtuelle Prozessor eines Einprozessor-Echtzeitgastsystems kann entweder fest einem bestimmten physischen Prozessor zugeordnet sein, oder (in Abbildung 4.13 gestrichelt dargestellt) im Rahmen einer statischen Planung wechselweise verschiedenen physischen Prozessoren zugewiesen sein. In diesem Fall ist der Scheduler für Einprozessor-Gastsysteme $(2, 2)$-

bzw $(1, 2)$-beschränkt.

- Jedem Echtzeit-Gastsystem ist ein Zeitfenster zugeordnet, das individuell für jeden virtuellen Prozessor dessen Zugang zum physischen Prozessor steuert. In der Regel wird dieses Zeitfenster genutzt, um den Rechenzeitkonsum jedes virtuellen Prozessors zu kontrollieren: Ein Echtzeitgastsystem kann durch einen periodischen unterbrechbaren Stellvertreterprozess repräsentiert werden, für den nach den in Abschnitt 3.1.3 angegebenen Methoden die Periodendauer Δp_{sv} und die Ausführungszeit Δe_{sv} ermittelt werden kann. Die Zeitfenster entziehen dem ihnen zugeordneten virtuellen Prozessor den physischen Prozessor, falls er innerhalb einer Periode Δp_{sv} länger als Δe_{sv} aktiv ist[1].

- Das Zeitfenster kann auch dazu genutzt werden, ein Coscheduling aller virtuellen Prozessoren einer virtuellen Maschine zu erreichen.

- Dem Stellvertreterprozess der Nicht-Echtzeitgastsysteme wird –ebenfalls durch ein Zeitfenster– eine Mindestauslastung U_{min} garantiert, um für die Gesamtheit der Nicht-Echtzeitprozesse einen festzulegenden durchschnittlichen Mindestfortschritt zusichern zu können.

- Darüber hinaus arbeitet der Stellvertreterprozess der Nicht-Echtzeitgastsysteme als „idle-"Prozess des $(2, 1)$-/$(1, 1)$-beschränkten Schedulers auf niedrigster Priorität. Auf diese Weise erhalten die Nicht-Echtzeitgastsysteme zusätzlich zur zugesicherten Mindestauslastung sämtliche für die Echtzeitgastsysteme eingeplante, dann aber nicht in Anspruch genommene Rechenzeit hinzu.

[1] was, wenn der Prozess sich entsprechend seiner Spezifikation verhält, niemals der Fall sein sollte.

Kapitel 5

Eine Scheduler-Infrastruktur für Virtual Machine Monitore

Kapitel 4 stellte am Ende die Anforderungen an einen VM-Scheduler zusammen, der die Koexistenz von Echtzeit- und Nicht-Echtzeitgastsystemen in getrennten virtuellen Maschinen im Kontext von Mehrprozessorsystemen ermöglicht. Außerdem wurde das Funktionsprinzip eines geeigneten VM-Schedulers skizziert. Dieses Kapitel beschreibt nun die Implementierung eines solchen Schedulers. Dabei wird bewusst kein Versuch unternommen, die Interessenskonflikte, die zwischen den verschiedenen Klassen von Gastsystemen zwangsläufig bestehen, automatisiert zu lösen: Wie am Beispiel des Konflikts zwischen zeit- und ereignisgesteuerten Systemen deutlich wurde, gibt es vielfach keine generische Lösung, d.h. es müssen immer Kompromisse eingegangen werden. Solche Kompromisse sind für jeden Anwendungsfall neu zu bewerten: Beispielsweise mag ein Verlust an Rechenleistung in der Größenordnung von 20% in Folge häufiger Umschaltungen für einen gegebenen Anwendungsfall vollkommen inakzeptabel sein, während er für einem anderen Fall tolerabel sein mag, da evtl. nur so eine bestehende Anforderung bezüglich der Reaktionszeit des Systems zu erfüllen ist. Jegliche Festlegung von Strategien innerhalb eines VM-Schedulers würde seinen Einsatzbereich auf die eine oder die andere Anwendung beschränken. Getreu dem Prinzip der Trennung von Strategie und Mechanismus ([LCC+75]) sollte deshalb ein Virtual Machine Monitor weitestmöglich frei von Strategien sein, aber es gestatten, verschiedene, alternative Strategien auf Basis der angebotenen Mechanismen umzusetzen.

Die eher statische Natur eingebetteter Systeme kann hier zum Vorteil genutzt werden: Da die Menge der Anwendungen und ihre Anforderungen bereits vorab, d.h. zum Zeitpunkt der Systemkonfiguration bekannt sind, können ihre Betriebsmittelzuteilungen statisch vorgenommen werden. Die zu entwerfende Infrastruktur

muss also die Konflikte nicht generell lösen, sondern sie muss lediglich einem Systemkonfigurator die nötigen Mittel an die Hand geben, damit dieser in Kenntnis einer konkreten Anwendung eine darauf optimal zugeschnittene Betriebsmittelverteilung realisieren kann.

5.1 Konzepte

5.1.1 Prozessorzuordnung

Der VM-Scheduler bildet die virtuellen Prozessoren von in der Regel mehreren virtuellen Maschinen auf einen oder mehrere physische Prozessoren der Rechnerhardware ab. Virtuelle Prozessoren, die verschiedenen physischen Prozessoren zugeordnet sind, können echt-parallel zueinander arbeiten. Virtuelle Prozessoren, die gleichen physischen Prozessoren zugeordnet sind, arbeiten dagegen nichtgleichzeitig, d.h. quasi-parallel.

Die in den virtuellen Maschinen arbeitenden Gastbetriebssysteme bilden wiederum diese virtuellen Prozessoren auf eine prinzipiell beliebige Anzahl von Prozessen ab. Hierbei sind Ein- und Mehrprozessorsysteme voneinander zu unterscheiden: Einprozessorbetriebssysteme nutzen nur einen einzigen virtuellen Prozessor. Damit können ihre Prozesse nur quasi-parallel arbeiten. Mehrprozessorbetriebssysteme sind dagegen in der Lage, mehrere virtuelle Prozessoren zu verwenden, und sie auf eine Anzahl von Prozessen zu verteilen. Die Anzahl der verwendeten virtuellen Prozessoren ist dabei auf Seiten des Gastsystems statisch konfiguriert, oder sie wird von diesem während der Startphase des Betriebssystems einmalig festgestellt. In jedem Fall ist diese Anzahl über die Laufzeit des Betriebssystems konstant und meist wesentlich niedriger als die zur Laufzeit veränderliche Anzahl der Prozesse.

Intern setzen Mehrprozessorbetriebssysteme voraus, dass ihre virtuellen Prozessoren echt-parallel arbeiten (siehe 3.3.1). Ist dies nicht gegeben, so kann es zu Leistungsverlusten (beispielsweise durch „lock holder preemption" (siehe 4.3.5)) oder sogar zu Deadlocks kommen. Solche Konstellationen sind somit nicht sinnvoll. Es kann davon ausgegangen werden, dass jedes Gastbetriebssystem, das in der Lage ist, mehrere Prozessoren zu betreiben, auch eine eigene Strategie besitzt, nach der es die Rechenkapazitäten dieser Prozessoren auf seine Anwenderprozesse verteilt. Für solche Systeme ist es bestenfalls gleichgültig, in der Regel aber kontraproduktiv, wenn eine weitere unterlagerte Schicht in Gestalt eines VM-Schedulers nochmals nach einer eigenen Strategie die endliche Anzahl virtueller Prozessoren auf eine u.U. andere, ebenfalls beschränkte Anzahl physischer Prozessoren abbildet. Daher sollte ein VM-Scheduler alle einer gemeinsamen virtuellen Maschine

zugewiesenen virtuellen Prozessoren stets fest an jeweils verschiedene physische Prozessoren binden. Damit muss die Anzahl virtueller Prozessoren pro virtueller Maschine stets kleiner oder gleich der Anzahl vorhandener physischer Prozessoren und während der Laufzeit der virtuellen Maschine konstant sein.

Am Beispiel des Konflikts zwischen zeit- und ereignisgesteuerten Anwendungen wurde in Abschnitt 2.2.6 bereits gezeigt, dass zeitliche Konkurrenzen zwischen verschiedenen virtuellen Maschinen auf Mehrprozessorsystemen zu lösen sind, indem die miteinander in Konflikt stehenden Anwendungen jeweils auf disjunkten Mengen von Prozessoren ausgeführt werden. Wird dies sichergestellt, so gibt es zwischen ihnen keinen gemeinsam genutzten Prozessor und damit keine Konflikte. Arbeiten die in Konflikt stehenden Prozesse unter einem gemeinsamen Mehrprozessorbetriebssystem, so ist es nach dem oben Gesagten Sache dieses Gastsystems, eine geeignete Strategie zur Zuordnung von Prozessen zu Prozessoren anzuwenden. Als Grundvoraussetzung benötigt es dazu allerdings die Garantie, dass die verwendeten virtuellen Prozessoren –wie oben ohnehin bereits gefordert– verschiedenen physischen Prozessoren zugeordnet sind, also parallel arbeiten. Arbeiten hingegen die in Konflikt stehenden Prozesse in getrennten virtuellen Maschinen, so ist es Aufgabe des VM-Schedulers, diesen virtuellen Maschinen disjunkte Mengen von Prozessoren zuzuweisen.

Die hier skizzierte Infrastruktur sieht vor, die Zuordnung von virtuellen zu physischen Prozessoren statisch zu konfigurieren: Für jeden virtuellen Prozessor gibt es eine *Affinitätsmenge*, d.h. eine Menge physischer Prozessoren, auf denen der betreffende virtuelle Prozessor potenziell ausgeführt werden kann. Diese wird einmalig zur Startzeit der virtuellen Maschine gewählt. Enthält diese Menge mehr als einen physischen Prozessor, so kann der VM-Scheduler, wenn er dem betreffenden virtuellen Prozessor startet, nach eigenen Kriterien einen Prozessor daraus auswählen. Die Strategie, nach der diese Auswahl getroffen wird, ist abhängig von der Klasse des Betriebssystems, das als Gast in der zugeordneten virtuellen Maschine arbeitet:

- Handelt es sich um ein *Echtzeitsystem*, so wird nach der „first-fit"-Methode vorgegangen, d.h. der erste gerade verfügbare Prozessor wird gewählt. Bei Echtzeitsystemen wird hiervon jedoch in den meisten Fällen kein Gebrauch gemacht, d.h. in der Regel wird die Konfiguration so gewählt, dass die Affinitätsmengen sämtlicher virtueller Prozessoren einer virtuellen Maschine jeweils genau einen (und zwar jeweils einen anderen) physischen Prozessor enthalten. Auf diese Weise wird die oben gestellte Forderung erfüllt, nach der alle virtuellen Prozessoren einer virtuellen Maschine stets fest an jeweils verschiedene physische Prozessoren gebunden sein sollen. Lediglich bei Einprozessor-Echtzeitsystemen kann es in einigen Fällen sinnvoll sein,

die Zuordnung des (einzigen) virtuellen Prozessors zu einem von mehreren alternativen physischen Prozessoren zu ermöglichen: Es gibt Situationen, in denen die Rechenzeitkontingente, die ein Einprozessorsystem zur Ausführung benötigt, zeitversetzt auf verschiedenen Prozessoren existieren. In solchen Fällen kann das Einprozessorsystem nur unter Inkaufnahme von Migrationen zwischen den Prozessoren eingeplant werden[1]. Dies entspricht dem in Abbildung 4.13 gestrichelt dargestellten Fall. Dabei ist, wie in 4.4 gesagt, der Scheduler für Einprozessor-Gastsysteme $(2, 2)$- bzw. $(1, 2)$-beschränkt.

- Bei *Nicht-Echtzeitsystemen* kann der Scheduler einen Lastausgleich vornehmen. Die Affinitätsmenge enthält daher für Nicht-Echtzeitsysteme entweder die Menge aller physischen Prozessoren, oder sie enthält wie bei Echtzeitsystemen jeweils nur einen (jeweils anderen) physischen Prozessor. Im ersten Fall ist der VM-Scheduler $(3, 3)$-beschränkt, d.h. er führt den Lastausgleich durch. Hiervon wird in der Regel bei Einprozessor-Gastsystemen Gebrauch gemacht. Im zweiten Fall ist der VM-Scheduler $(3, 1)$-beschränkt, d.h. der Lastausgleich muss in diesem Fall vom Gastsystem übernommen werden. Diese Variante wird bei Mehrprozessor-Gastsystemen verwendet.

Die Wahl der Affinitätsmenge erfüllt somit zwei Anforderungen: zum Einen kann dadurch für Einprozessorsysteme die Strategie vorgegeben werden, nach der die Rechenlast auf verschiedene physische Prozessoren verteilt wird, und zum Anderen kann damit die geforderte Entkopplung, beispielsweise zwischen Zeit- und Ereignissteuerung, realisiert werden, indem die Affinitätsmengen der zueinander in Konkurrenz stehenden virtuellen Prozessoren so gewählt werden, dass sie disjunkt sind.

Durch ungünstig gewählte Affinitätsmengen können auch unvorteilhafte Situationen entstehen. So können sowohl notwendige Migrationen verhindert, als auch überflüssige Migrationen ermöglicht werden. Es liegt allein in der Verantwortung des Systemkonfigurators, die Affinitätsmengen nach den obengenannten Regeln sinnvoll zu wählen. Aus diesem Grund sollte eine virtuelle Maschine auch maximal nur lesenden Zugriff auf die eigenen Affinitätsmengen besitzen.

5.1.2 Prioritätssteuerung

Virtuelle Maschinen, die Echtzeit-Anwendungen enthalten, lassen sich gegenüber dem VM-Scheduler durch Periodendauern Δp_{sv} und Ausführungszeiten Δe_{sv}

[1] wobei die dadurch entstehenden Migrationsverluste in die Planung bereits miteinbezogen werden können

charakterisieren. Diese Größen können bei Kenntnis der in der virtuellen Maschine ausgeführten Prozesse und des zu ihrer Planung eingesetzten lokalen Verfahrens ermittelt werden (vgl.3.1.3). Der VM-Scheduler muss nun sicherstellen, dass die virtuelle Maschine ihrerseits in jedem Zeitintervall der Länge Δp_{sv} insgesamt für eine Zeitdauer Δe_{sv} ausgeführt wird. Ist das gegeben, so kann auch das Gastsystem seinen Anwenderprozessen eine fristgerechte Ausführung garantieren.

Die Planung auf Ebene des VM-Schedulers kann damit nach einem Verfahren für periodische, unterbrechbare Prozesse vorgehen. Sowohl für das EDF- als auch für das RMS-Verfahren wurden in 2.2.7 Vorschriften zum Entscheiden der Planbarkeit einer durch Periodendauern und Ausführungszeiten spezifizierten Menge unterbrechbarer periodischer Prozesse angegeben. Beide Planungsverfahren lassen sich mit Hilfe eines prioritätsgesteuerten Schedulers umsetzen: Für RMS genügen statische Prioritäten, während für EDF auftragsbezogen dynamische Prioritäten erforderlich sind. Der hier skizzierte Scheduler verwendet priortätsgesteuerte Warteschlangen, wobei vereinfachend davon ausgegangen wird, dass jede Prioritätsstufe höchstens einmal vorkommt, d.h. der Fall, dass mehrere virtuelle Prozessoren gleichzeitig dieselbe Prioritätsstufe besitzen, wird hier ausgeschlossen. Diese Vereinfachung ist sowohl für EDF als auch RMS zulässig: Sollte bei Anwendung eines dieser beiden Verfahren mehreren Prozessen die gleiche Priorität zugewiesen werden, so ist die Ausführungsreihenfolge dieser Prozesse willkürlich wählbar, ohne dass dabei eine Fristverletzung auftritt. Somit sollte sich für alle planbaren Situationen eine eindeutige Prioritätenzuordnung bestimmen lassen.

Da nach dem in 5.1.1 Gesagten der VM-Scheduler entweder keine Migrationen oder allenfalls eingeschränkte Migrationen zulässt (siehe 4.2.1), gibt es bei dem hier skizzierten VM-Scheduler eine prioritätsgesteuerte Warteschlange für jeden physischen Prozessor. Die Prozesse (d.h. die virtuellen Prozessoren) werden zu ihren jeweiligen Bereitzeiten entsprechend ihrer Priorität dort eingereiht.

Echtzeitgastsysteme können, wenn sie keinen Rechenzeitbedarf haben, oder wenn die innerhalb einer Periode zu erfüllende Aufgabe in kürzerer als der zugewiesenen Zeit abgeschlossen wurde, blockieren. Die dabei anfallende, nicht genutzte Rechenzeit wird so für andere Aktivitäten mit niedrigerer Priorität frei. Befindet sich kein rechenbereiter virtueller Prozessor in der Warteschlange eines physischen Prozessors, so übergibt der prioritätsgesteuerte Scheduler die Kontrolle an den „fairen" Scheduler, der die Nicht-Echtzeitgastsysteme steuert. Diese erhalten somit sämtliche Rechenzeit, die von keinem der Echtzeitgastsysteme genutzt wird. Sie arbeiten damit gewissermaßen als „idle-"Prozess des prioritätsgesteuerten Schedulers und übernehmen so die Rolle des „Konsumenten" ungenutzter Rechenkapazitäten.

Darüber hinaus besteht die optionale Möglichkeit, dem „fairen" Scheduler, bzw. den von ihm vertretenen Nicht-Echtzeitanwendungen eine gewisse Mindestmenge an Rechenkapazität zuzusichern, um so ein „Verhungern" auch für den (unwahrscheinlichen) Fall zu verhindern, dass alle Echtzeit-Anwendungen alle ihnen zugesicherten Rechenkapazitäten vollständig ausschöpfen (s. 5.1.3).

Ähnlich wie bereits bei den Affinitätsmengen können auch durch falsch konfigurierte Zeitfenster Schäden angerichtet werden, die über die Grenzen einzelner virtuellen Maschine hinaus Wirkung zeigen. Aus diesem Grund obliegt es auch hier dem Systemkonfigurator, sinnvolle (d.h. dem Zeitbedarf der des Stellvertreterprozesses angemessene) Zeitfenster an die virtuellen Maschinen zu vergeben. Eine virtuelle Maschine darf daher selbst maximal nur lesenden Zugriff ihr Zeitfenster besitzen.

5.1.3 Zeitsteuerung

Die Rechenkapazitäten, die ein Echtzeitgastsystem benötigt, um seine Prozesse fristgerecht auszuführen, sind begrenzt. Wie bereits ausgeführt, können sie durch Periodendauern Δp_{sv} und Ausführungszeiten Δe_{sv} beschrieben werden. Aufgrund dieser Parameter werden den Gastsystemen, wie zuvor beschrieben, nach dem RMS- oder dem EDF-Verfahren Prioritäten zugewiesen. Beispielsweise werden beim RMS-Verfahren die Prioritäten umgekehrt zur Größe der Periodendauer Δp_{sv} gewählt, d.h., hat ein Gastsystem eine kurze Periodendauer, so erhält es eine hohe Priorität. Ohne besondere Schutzmaßnahmen versetzt diese hohe Priorität das Gastsystem technisch in die Lage, alle anderen Gastsysteme, die eine niedrigere Priorität besitzen, beliebig lange zu blockieren. Solange alle Gastsysteme korrekt arbeiten, kann eine solche Situation nicht eintreten, aber in einer Virtualisierungsumgebung müssen auch die Folgen von fehlerhaftem Verhalten eines Gastsystems sicher auf die virtuelle Maschine dieses Gastsystems beschränkt bleiben (vgl. dazu auch Kapitel 1 (Einführung), Stichwort „Fehlerlokalität"). Um diese „Fehlerlokalität" gewährleisten zu können, wird jedem virtuellen Prozessor ein individuelles periodisches Zeitfenster zugeteilt. Dieses Zeitfenster beschränkt die dem virtuellen Prozessor in jedem Zeitintervall der Dauer Δp_{sv} zugewiesene Rechenzeit auf maximal Δe_{sv}. Versucht ein virtueller Prozessor, diese Grenze zu überschreiten, so wird er unterbrochen und für den Rest der laufenden Periode suspendiert. Auf diese Weise wird sichergestellt, dass auch bei (böswillig oder unabsichtlich) vorhandenen Fehlfunktionen einzelner Gastsysteme die Gültigkeit des Ausführungsplans des VM-Schedulers erhalten bleibt. Die Folgen des Fehlers bleiben auf diejenige virtuelle Maschine beschränkt, die ihn verursacht hat.

Die Technik der Zeitfenster wird auch genutzt, um den „fairen" Scheduler mit

einer frei wählbaren Periodendauer Δp_{sv} und Ausführungszeit Δe_{sv} zu auszustatten, sodass auch er in gleicher Weise wie die Echtzeit-Gastsysteme durch den RMS- bzw. EDF-Scheduler eingeplant werden kann. So wird die Mindestauslastung $U_{min} = \frac{\Delta e_{sv}}{\Delta p_{sv}}$ sichergestellt. Im Unterschied zu den Echtzeit-Gastsystemen ist hier das Ausschöpfen des Zeitlimits kein Zeichen einer Fehlfunktion, sondern normales Verhalten: Da die Nicht-Echtzeitgastsysteme gierig sind, trifft dies auch auf ihren Stellvertreter zu, d.h. dieser konsumiert sämtliche ihm zugängliche Rechenkapazität. Das Zeitfenster wird hier also dazu genutzt, diesen Konsum zu begrenzen.

5.1.4 Der „faire" Scheduler

Der „faire" Scheduler hat die Aufgabe, die Gesamtheit der Nicht-Echtzeitgastsysteme zu einem Stellvertreterprozess zu kapseln. Diesem Stellvertreterprozess wird vom prioritätsgesteuerten Scheduler sämtliche Rechenzeit zugeteilt, die von den virtuellen Maschinen mit Echtzeit-Anwendungen nicht konsumiert wurde. Zusätzlich kann ein festes periodisches Zeitkontingent zugeteilt werden, das eine minimale Auslastung sicherstellt. Die Gesamtheit dieser zugeteilten Rechenkapazitäten wird durch den „fairen" Scheduler gleichmäßig über alle Nicht-Echtzeitgastsysteme verteilt. Nach 4.4 arbeitet der Scheduler für Einprozessorgastsysteme $(3, 3)$- und für Mehrprozessorgastsysteme $(3, 1)$-beschränkt, was über die Konfiguration der Affinitätsmengen der virtuellen Prozessoren gesteuert wird (s.o.).

Der „faire" Scheduler verwaltet ausschließlich Betriebsmittel, die für die Echtzeitverarbeitung nicht (oder nicht mehr) relevant sind. Seine Arbeitsweise ist daher ohne Einfluss auf das Zeitverhalten von Echtzeitanwendungen. Daher sollen hier bewusst keine Festlegungen getroffen werden: Jedes arbeitserhaltende Verfahren, das eine hinreichend faire Verteilung bietet, ist hier anwendbar. Beispielsweise könnte die Zuteilung nach dem Round-Robin-Verfahren erfolgen, oder es könnte nach einem der Verfahren zur anteiligen Zuordnung vorgegangen werden. Die Verteilung der Rechenzeit auf die virtuellen Prozessoren kann über Gewichte gesteuert werden, eventuell werden auch Heuristiken benutzt, um die Gewichte dynamisch an die jeweilige Lastsituation anzupassen.

Da die gleichmäßige Verteilung von Rechenleistung ohne Echtzeitanspruch ohnehin die typische Betriebssituation der meisten heute existierenden Virtualisierungslösungen darstellt, kann davon ausgegangen werden, dass die in Produkten wie VMware oder Xen integrierten Scheduler ausgereift und für diesen Zweck bestens geeignet sind.

5.1.5 Coscheduling

Durch die zuvor in 5.1.1 beschriebene statische Prozessorzuordnung über Affinitätsmengen ist auch der spezielle Fall einer 1:1 Zuordnung von virtuellen zu physischen Prozessoren innerhalb einer virtuellen Maschine realisierbar, d.h. die virtuellen Prozessoren, die die virtuelle Maschine „sieht", konkurrieren niemals um denselben physischen Prozessor. Werden überdies diesen virtuellen Prozessoren, wie zuvor in 5.1.3 beschrieben, gleiche Zeitfenster mit gleicher, fester Startzeit und mit hinreichend hoher Priorität zugewiesen, so kann für die betreffende virtuelle Maschine Coscheduling garantiert werden, d.h. alle ihre Prozessoren arbeiten immer gleichzeitig, niemals zeitversetzt.

Auch hier obliegt es dem Systemkonfigurator, die von der Scheduler-Infrastruktur angebotenen Mechanismen (Priorität, Zeitfenster und Affinitätsmengen) zur Durchsetzung einer Strategie (hier: Coscheduling) zu nutzen.

5.2 Entwurf

Dieser Abschnitt beschreibt die Konstruktion der skizzierten Scheduler-Infrastruktur. Es werden zunächst die benötigten Datenstrukturen und anschließend die erforderlichen Funktionen beschrieben.

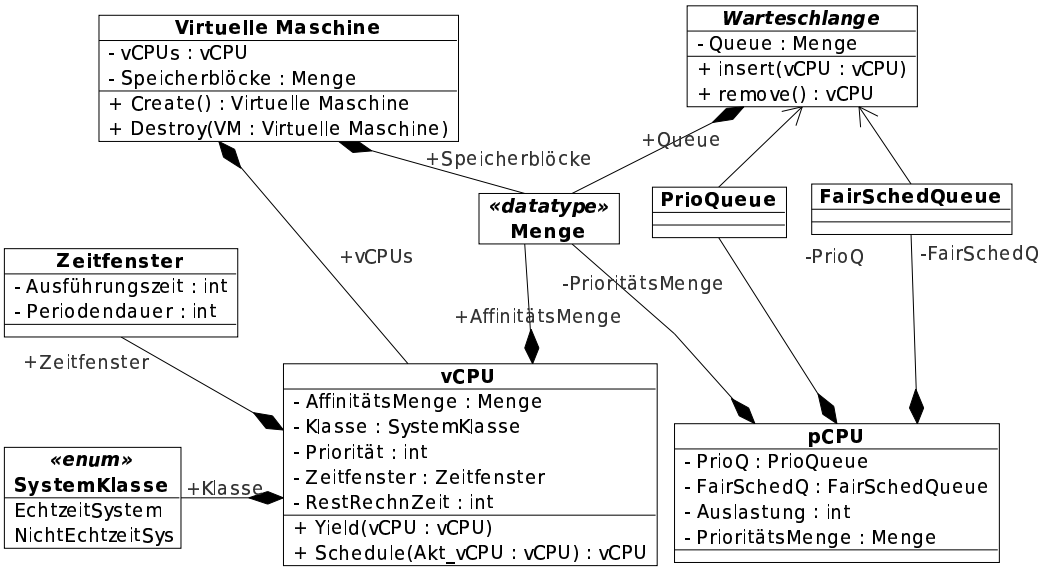

Abbildung 5.1: Klassendiagramm des VM-Schedulers.

Abbildung 5.1 zeigt die Objekte des Schedulers in einem Klassendiagramm.

5.2.1 Warteschlangen

Innerhalb des VM-Schedulers dienen Warteschlangen dazu, Verweise auf virtuelle Prozessoren zu speichern und in eine bestimmte Reihenfolge zu bringen. Die Methode `insert()` speichert ein Objekt (hier: einen virtuellen Prozessor), die Methode `remove()` entnimmt das vorderste Objekt aus der Warteschlange. Wie in 4.4 gezeigt, kommen zwei verschiedene Scheduler zum Einsatz. Dementsprechend gibt es zwei verschiedene Wartedisziplinen:

1. Prioritäts-Warteschlange: In dieser Klasse von Warteschlangen werden virtuelle Prozessoren gehalten, deren Gastsysteme Echtzeitsysteme sind. Es gibt endlich viele Prioritätsstufen (z.B.: 256). Die `remove()`-Methode liefert hier jeweils den höchst-priorisierten virtuellen Prozessor aus der Warteschlange. Da jede Prioritätstufe nur von maximal einem virtuellen Prozessor verwendet werden kann, ist diese Auswahl stets eindeutig.

2. Nicht-Echtzeit-Warteschlange: Hier werden virtuelle Prozessoren eingereiht, deren Gastsysteme keine Echtzeitanforderungen haben. Die Gesamtheit dieser virtuellen Prozessoren stellt aus Sicht der Prioritäts-Warteschlange deren „idle-"Prozess dar, d.h. alle Rechenkapazität, die von den höher priorisierten virtuellen Prozessoren nicht genutzt wurde, fällt dieser Ebene zu. Der „faire" Scheduler arbeitet auf dieser Klasse von Warteschlangen. Ihre Organisation hängt stark vom verwendeten Algorithmus des „fairen" Schedulers ab, sodass hier keine konkreten Angaben dazu möglich sind. In den meisten Fällen ist allerdings eine verkette Liste die geeignete Form der Darstellung dieser Warteschlange.

5.2.2 Physische Prozessoren

In Kapitel 4 wurde begründet, warum für nahezu alle sinnvollen Betriebsfälle eine feste Bindung von virtuellen zu physischen Prozessoren bestehen muss. Einzige Ausnahme sind hier Nicht-Echtzeit-Einprozessorsysteme, für die es in vielen Fällen sinnvoll ist, ihren virtuellen Prozessor dynamisch wechselnden physischen Prozessoren zuzuordnen. In der Mehrzahl der hier betrachteten Konfigurationen kommt als Nicht-Echtzeitsystem Linux, ein Mehrprozessorbetriebssystem zum Einsatz. Ein Nicht-Echtzeit-Einprozessorsystem wird also eine seltene Ausnahme sein. Die feste Zuordnung zwischen physischen und virtuellen Prozessoren ist damit der eindeutig häufigere Betriebsfall, deshalb wird die Infrastruktur auf diesen Fall optimiert und verwendet prozessorlokale Warteschlangen, d.h. jeder

physische Prozessor besitzt eigene Warteschlangen. Ein Verlagern eines virtuellen Prozessors zu einem anderen physischen Prozessor wird dadurch nicht ausgeschlossen, sondern lediglich aufwändiger gemacht: Es erfordert das ausdrückliche Austragen des Prozessors aus einer Warteschlange und das Eintragen in eine andere.

Da es zwei Klassen von Warteschlangen gibt (s.o.) besitzt jeder physische Prozessor zwei Warteschlangen: eine Prioritäts- und eine Nicht-Echtzeit-Warteschlange.

Beim Erzeugen von virtuellen Prozessoren (s.u.) muss ein Planbarkeitstest durchgeführt werden, um zu entscheiden, ob die von dem neuen virtuellen Prozessor benötigten Betriebsmittel verfügbar sind. Dazu muss über die aktuelle Auslastung jedes physischen Prozessors Buch geführt werden. Zu diesem Zweck besitzt jeder physische Prozessor einen Auslastungszähler, dessen Wert sich nach Gleichung (2.3) (siehe 2.2.1) aus den Periodendauern und Ausführungszeiten der Zeitfenster aller diesem physischen Prozessor aktuell zugeordneten virtuellen Prozessoren berechnet. Dieser Zähler wird nach Art eines Kontostandes verwaltet, d.h. für jeden neu hinzukommenden virtuellen Prozessor wird er um dessen Auslastungsbeitrag erhöht (natürlich erst nachdem der Planbarkeitstest positiv entschieden wurde – s.u.), für jeden verworfenen virtuellen Prozessor wird er um dessen Auslastungsbeitrag verringert. Als Initialwert der Auslastung kann ein bei der Systemkonfiguration festzulegender Wert U_{min} angegeben werden. Auf diese Weise wird, wie in 4.1.4 gezeigt, ein Anteil der Rechenkapazität für die Verwendung durch Nicht-Echtzeitsysteme reserviert, um deren „Verhungern" sicher ausschließen zu können.

Da die Scheduler-Infrastruktur verlangt, dass alle virtuellen Prozessoren, die Echtzeit-Anforderungen zu genügen haben, eine eigene, exklusiv verwendete Priorität besitzen (Dies vereinfacht die Struktur der Warteschlangen), muss für jede Prioritäts-Warteschlange Buch über die Vergabe von Prioritäten geführt werden. Hierzu existiert für jeden physischen Prozessor eine Prioritätsmenge der jede genutzte Priorität hinzugefügt wird. Versuche, eine Priorität mehrfach zu vergeben, können so erkannt und zurückgewiesen werden.

5.2.3 Virtuelle Prozessoren

Die virtuellen Prozessoren stellen aus der Sicht eines VM-Schedulers Prozesse dar. Dementsprechend werden sie durch Datenstrukturen beschrieben, die Ähnlichkeit mit den Prozessdeskriptoren klassischer Betriebssysteme haben. Sie enthalten Elemente zum Retten von Prozessorregistern, Verweiszeiger zum Einhängen der Struktur in verschiedene Warteschlangen, usw. Für die Scheduler-Infrastruktur sind folgende Attribute von Bedeutung:

- Affinitätsmenge
- Klasse: Echtzeit oder Nicht-Echtzeit
- Priorität
- Zeitfenster
- Verbleibende Rechenzeit

Die Affinitätsmenge legt die Menge der physischen Prozessoren fest, denen der zugehörige virtuelle Prozessor zugeordnet werden kann. In den meisten Fällen enthält diese Menge nur einen physischen Prozessor, d.h. die Zuordnung ist eindeutig, doch sind auch andere Konstellationen möglich.

Virtuelle Prozessoren, die Nicht-Echtzeit Gastsysteme ausführen, arbeiten konzeptionell gemeinsam auf niedrigster Prioritätsstufe und unterliegen gemeinsamen dem „fairen" Scheduler, der die anfallende Rechenzeit gleichmäßig über alle seine virtuellen Prozessoren verteilt. Die Angabe einer Priorität wäre für sie redundant. Echtzeitgastsysteme besitzen dagegen eine individuelle Priorität, die den Vorrang zwischen in Konflikt stehenden virtuellen Prozessoren regelt. Die Klasse eines virtuellen Prozessors gibt an, ob er für die Ausführung von Echtzeit- oder Nicht-Echtzeit-Anwendungen verwendet wird. Abhängig von dieser Klasse erfolgt das Einfügen des Prozessors entweder in die Prioritäts-Warteschlange oder in die Nicht-Echtzeit-Warteschlange des über die Affinitätsmenge zugeordneten virtuellen Prozessors. Die Priorität ist dabei nur relevant, falls der virtuelle Prozessor in die Prioritäts-Warteschlange eingefügt wird (falls also die Klasse den Wert „Echtzeit" hat).

Jedem virtuellen Prozessor ist ein periodisches Zeitfenster zugewiesen, das durch Periodendauer und Ausführungszeit je Periode bestimmt ist. Für das laufende Zeitfenster wird ein Zähler für die noch verfügbare Rechenzeit mitgeführt. Wird er Null, so wird der betreffende virtuelle Prozessor bis zum Ende des laufenden Zyklus blockiert. Der Zähler wird zu Beginn jeder neuen Periode auf den Maximalwert zurückgesetzt.

Einem virtuellen Prozessor sind die Methoden `Schedule()` und `Yield` zugeordnet, die in den folgenden Abschnitten beschrieben werden.

5.2.4 `Schedule()` – Auswahl des nächsten virtuellen Prozessors

Dies ist die zentrale Funktion des VM-Schedulers. Ihre Aufgabe ist es, den laufenden virtuellen Prozessor zu blockieren, den nächsten virtuellen Prozessor auszu-

wählen und ihn zu starten. Folgende Ereignisse führen zum Aufruf der Schedule-Funktion:

- Ablauf des Zeitfensters des laufenden virtuellen Prozessors
- Beginn eines neuen Zeitfensters
- Blockieren des laufenden virtuellen Prozessors
- Eintritt eines Ereignisses, durch das ein bisher blockierter virtueller Prozessor rechenwillig wurde
- Freiwillige Abgabe des physischen Prozessors (Yield)

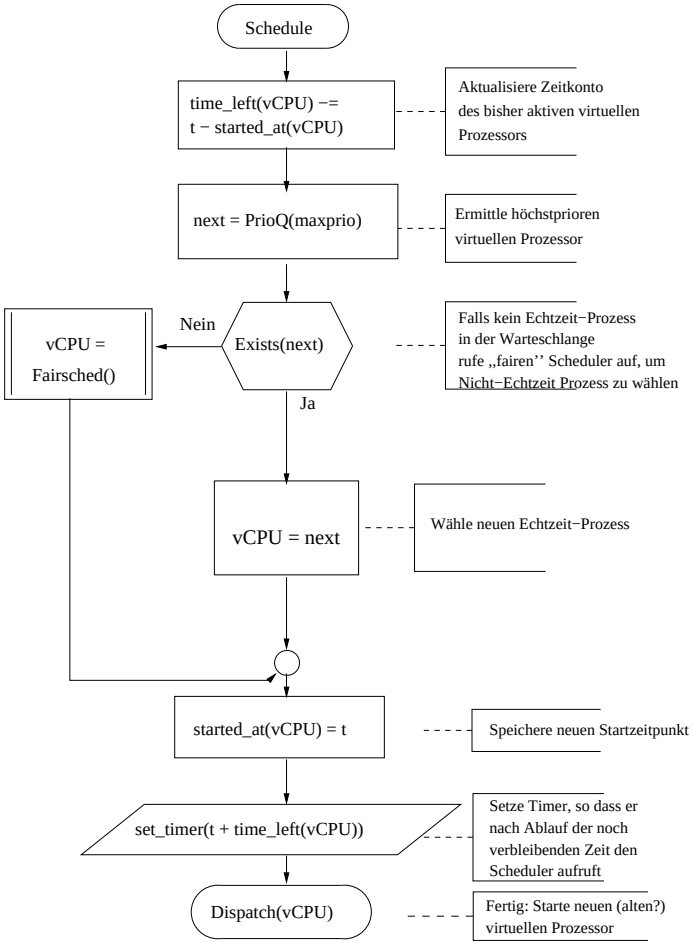

Abbildung 5.2: Flussdiagramm der Schedule-Funktion.

Abbildung 5.2 zeigt ein Flussdiagramm der Schedule-Funktion. In allen Fällen wird der betroffene physische Prozessor dem laufenden virtuellen Prozessor entzogen, und es wird ein neuer virtueller Prozessor zur Aktivierung ausgewählt. Dabei kann die Wahl prinzipiell auch wieder auf denselben virtuellen Prozessor fallen, der zuvor aktiv war. In den meisten Fällen findet jedoch ein Wechsel statt.

Beim Eintritt in die Funktion wird das Zeitkonto des bisher aktiven virtuellen Prozessors aktualisiert, d.h. sein Zähler für die verbleibende Rechenzeit wird um die seit seinem letzten Start verstrichene[1] Zeit vermindert. Erreicht der Zähler dabei den Wert Null, d.h. hat der virtuelle Prozessor alle ihm in der aktuellen Periode zustehende Zeit verbraucht, so bleibt er bis zum Beginn seiner nächsten Periode blockiert.

Anschließend wird zunächst versucht, einen rechenbereiten virtuellen Prozessor aus der Prioritäts-Warteschlange zu entnehmen. Gibt es mehr als einen rechenbereiten virtuellen Prozessor, so liefert die Prioritäts-Warteschlange denjenigen mit der höchsten Priorität. Findet sich ein rechenbereiter virtueller Prozessor, so wird dieser gestartet, d.h. er erhält den physischen Prozessor.

Gibt es in der Prioritäts-Warteschlange aktuell keinen rechenwilligen virtuellen Prozessor, so wird der „faire"Scheduler aufgerufen, der aus der Menge der ihm unterstellten virtuellen Prozessoren mit Nicht-Echtzeit-Anwendungen nach seinen Regeln (d.h. in der Regel anteilige Zuteilung) einen zur Ausführung auswählt.

Neben dem zu startenden virtuellen Prozessor ermittelt der Scheduler auch die Zeitdauer, für die dieser maximal aktiv bleiben darf. Diese Zeit ist gleich dem aktuellen Stand seines Zeitkontos. Der Scheduler initialisiert einen Timer so, dass nach Ablauf dieser Zeitspanne ein Interrupt ausgelöst wird, der wiederum zum Aufruf des Schedulers führt. Auf diese Weise werden die von den virtuellen Prozessoren konsumierten Zeitmengen überwacht. Keiner von ihnen kann innerhalb seiner Periode mehr als die zugewiesene Menge an Rechenzeit konsumieren.

5.2.5 `Yield()` – Zeitscheibe aufgeben

Durch Aufruf der Yield-Funktion signalisiert ein virtueller Prozessor seine Bereitschaft, den physischen Prozessor, den er derzeit nutzt, vorübergehend freizugeben. Die Auswirkungen dieses Aufrufs hängen von der Klasse des aufrufenden virtuellen Prozessors ab:

- Bei Echtzeitgastsystemen wird das aktuelle Zeitfenster vorzeitig beendet. Der virtuelle Prozessor wird bis zum Beginn seines nächsten Zeitfensters

[1] Um diese verstrichene Zeit ermitteln zu können, wird bei jedem Start eines virtuellen Prozessors die absolute Zeit in dessen Verwaltungsstruktur gespeichert.

blockiert. Die im laufenden Zeitfenster nicht in Anspruch genommene Restrechenzeit verfällt aus der Sicht des Aufrufers. Sie fällt der Gesamtheit der Nicht-Echtzeitgastsysteme zu.

- Virtuelle Prozessoren, die Nicht-Echtzeit-Gastsysteme ausführen, die also dem „fairen' Scheduler unterstellt sind, geben ihren physischen Prozessor ab, bleiben aber rechenwillig. Der „faire" Scheduler entscheidet hier über die Vorgehensweise. In der Regel wird der aufrufende virtuelle Prozessor ans Ende der Warteschlange gestellt, sodass er erst wieder aktiviert wird, nachdem alle anderen rechenwilligen Nicht-Echtzeit-Gastsysteme reihum aktiviert worden sind

5.2.6 Virtuelle Maschinen

Für den VM-Scheduler sind virtuelle Maschinen einfach Mengen virtueller Prozessoren, sowie Mengen von „räumlichen" Betriebsmitteln (z.B. Speicherblöcke), die von diesen virtuellen Prozessoren gemeinsam zu nutzen sind. An dieser Stelle sind allerdings nur die Aspekte der Prozessplanung und -Ausführung von Interesse, deshalb werden diese „räumlichen" Betriebsmittel hier nicht weiter betrachtet.

Wegen des angesprochenen Konzepts der Trennung von Strategie und Mechanismus macht der Scheduler bewusst keine Annahmen über Beziehungen zwischen virtuellen Prozessoren, die derselben virtuellen Maschine angehörigen. Sollen z.B. virtuelle Prozessoren einer gemeinsamen virtuellen Maschine stets gleichzeitig auf verschiedenen Prozessoren aktiv sein (das entspräche dem „Coscheduling"), so muss dies durch geeignetes Festlegen der Affinitätsmengen und der Zeitfenster der einzelnen virtuellen Prozessoren erreicht werden. Allein aus der Zugehörigkeit zu einer gemeinsamen virtuellen Maschine ergeben sich keine solchen Beziehungen.

Virtuelle Maschinen können mit Hilfe der Funktionen `Create()` und `Destroy()` dynamisch erzeugt bzw. verworfen werden.

5.2.7 `Create()` und `Destroy()` – Erzeugen und Verwerfen von virtuellen Maschinen

Diese Operationen sind nur einem Systemkonfigurator zugänglich. Je nach Implementierung kann es dazu beispielsweise eine ausgezeichnete virtuelle Maschine geben (bei Xen wäre dies die „Domain0"), für die diese Schnittstelle aufrufbar ist.

Des Erzeugen einer virtuellen Maschine mit `Create()` beinhaltet das Erzeugen aller ihrer virtuellen Prozessoren. Dazu müssen deren Parameter, also die

bereits besprochenen Werte für Affinitätsmenge, Klasse, Priorität, Periodendauer und Ausführungszeit des Zeitfensters angegeben werden. Die Betriebsmittel der virtuellen Prozessoren werden bei ihrer Erzeugung alloziert. Dies bezieht sich sowohl auf „räumliche" (d.h. Speicherplatz) als auch auf „zeitliche" Betriebsmittel. Sind nicht genügend freie Betriebsmittel vorhanden, so schlägt die Erzeugung fehl.

Die „räumlichen" Betriebsmittel sollen hier, wie oben bereits begründet, nicht ausführlich betrachtet werden. Im Wesentlichen geht es darum, eine ausreichende Menge an Speicherplatz zu beschaffen, in dem die Datenstrukturen der virtuellen Prozessoren angelegt werden. Die Datenstrukturen zur Verwaltung physischer Prozessoren werden im Gegensatz zu den virtuellen Prozessoren bereits zur Startzeit des Virtual Machine Monitors angelegt, sind also bereits vorhanden, wenn virtuelle Maschinen erzeugt werden.

Ist eine virtuelle Maschine aufgrund ihrer Klasse als Echtzeitsystem gekennzeichnet, so wird zunächst geprüft, ob die gewählte Priorität bereits von einem anderen virtuellen Prozessor genutzt wird. Ist dies der Fall, so schlägt die Erzeugung fehl. Falls die Affinitätsmenge des virtuellen Prozessors mehr als einen physischen Prozessor enthält, so werden diese Prüfungen für alle physischen Prozessoren in der Menge durchgeführt.

Anschließend wird die Gesamtauslastung des physischen Prozessors betrachtet. Dazu wird der aus Ausführungszeit und Periodendauer zu berechnende Auslastungsbeitrag des neuen virtuellen Prozessors zum Auslastungszähler des physischen Prozessors hinzuaddiert, und es wird geprüft, ob das jeweilige Planbarkeitskriterium des Schedulers (d.h. eine der Gleichungen (2.6), (2.8), (4.24) oder (4.23)) erfüllt ist. Ist dies der Fall, so wird der virtuelle Prozessor zugelassen und der Auslastungszähler des physischen Prozessors wird entsprechend erhöht.

Man beachte, dass hier keine Prüfung der Korrektheit des für die jeweiligen virtuellen Prozessoren angegebenen Zeitfensters vorgenommen wird: Es ist Aufgabe des Systemkonfigurators, eine lokale Planbarkeitsanalyse des Gastsystems durchzuführen. Die Vorgehensweise hierzu wurde in Abschnitt 3.1.3 ausführlich beschrieben. Sie liefert als Ergebnis die Periodendauer Δp_{sv} und die Ausführungszeit Δe_{sv}, die das hier anzusetzende Zeitfenster bestimmen.

Umgekehrt zu `Create()` führt das Verwerfen einer virtuellen Maschine mit `Destroy()` zum Verwerfen aller zugehörigen virtuellen Prozessoren und zur Freigabe der von diesen belegten Betriebsmittel.

5.3 Implementierung

Um die in Abschnitt 5.1 beschriebenen Konzepte praktisch zu erproben, wurde eine eigene Scheduler-Infrastruktur für den Virtual Machine Monitor Xen geschaffen. Diese Infrastruktur weicht in ihrer derzeitigen Implementierung in Teilen von der in 5.2 vorgestellten Struktur ab, da sie zeitlich vor dieser Ausarbeitung entstanden ist. Die Unterschiede zum Entwurf sind teilweise darauf zurückzuführen, dass einige Konzepte sich während der praktischen Erprobung als verbesserungsfähig erwiesen haben. Damit kann die in 5.2 beschriebene Infrastruktur als Weiterentwicklung der in diesem Abschnitt erörterten Implementierung angesehen werden. Nichtsdestotrotz sollte auch diese Struktur, da sie als Testumgebung für viele der im Rahmen dieser Arbeit durchgeführten Experimente gedient hat, hier vorgestellt werden.

5.3.1 Xen als Testumgebung

Der Virtual Machine Monitor Xen ([BDF+03]) zielt eindeutig *nicht* auf eingebettete Systeme ab. Er besitzt etliche Merkmale, die für einen Einsatz im Umfeld eingebetteter Systeme überflüssig sind:

- *Live Migration*: Verschieben von Gästen von einem Wirts-Rechner auf einen anderen, auch im laufenden Betrieb, innerhalb einiger Millisekunden.
- *Virtuelle Netzwerke*: Die virtuellen Maschinen können virtuell miteinander vernetzt werden.
- *Management Funktionalität*: Schnittstelle für Web-basierte Administration und Überwachung, dynamisches Vergrößern oder Verkleinern der Speicherzuteilung der virtuellen Maschinen.

All diese Funktionalitäten finden ihren Niederschlag im Code-Umfang des Virtual Machine Monitors: Aktuelle Versionen umfassen, je nach Zählweise, zwischen 150.000 und 200.000 Zeilen Quellcode ([Fra08]), der Laufzeit-Speicherbedarf liegt bei ca. 16MB ([Hei09]). Hinzu kommt, dass Xen sämtliche Ein/Ausgabe sowie seine eigene dynamische Steuerung an ein Linux-System delegiert, das hierzu in einer speziellen, privilegierten virtuellen Maschine (der sogenannten „Domain0") ausgeführt werden muss. Allein der Betriebssystemkern dieses Linux-Systems umfasst mehrere Millionen Zeilen Quellcode und, da die Steuerungssoftware in der Programmiersprache Python abgefasst ist, ist darüber hinaus noch ein vollständiger Python-Interpreter notwendig.

Der hieraus resultierende Betriebsmittelbedarf eines Xen-Systems sprengt die Grenzen der meisten eingebetteten Systeme bei weitem, und an eine erschöpfende Validierung der gesamten „Trusted Code Base", wie sie im sicherheitkritischen Umfeld gefordert ist, ist angesichts mehrerer Millionen Zeilen Quellcode nicht zu denken.

Andererseits besitzt Xen einige Eigenschaften, die es als Testbett zur Erprobung neuer Planungsstrategien prädestinieren:

- *Quelloffenheit*: Der gesamte Quellode des Virtual Machine Monitors und der zugehörigen Werkzeuge ist offengelegt und frei verfügbar. Er wird von einer großen, weltweit verteilten Entwicklergemeinde gepflegt und weiterentwickelt. Viele dieser Entwickler sind über die Entwickler-Mailingliste ansprechbar.

- *Mehrprozessorfähigkeit*: Xen unterstützt bereits SMP-Architekturen, die auf der IA-32 Prozessorfamilie basieren.

- *Plug-in-Schnittstelle für Scheduler*: Xen besitzt eine interne Schnittstelle für den Einbau alternativer Scheduler. Diese Schnittstelle erlaubt es, mehrere, jeweils durch zwölf Funktionen zu definierende Scheduler-Module einzubinden. Aus der Liste der verfügbaren Scheduler wird beim Start des Systems durch einen Aufrufparameter derjenige Scheduler ausgewählt, der zum Einsatz kommen soll.

Obwohl also Xen für den Einsatz in eingebetteten Systemen kaum jemals in Frage kommen dürfte, ist es vor allem wegen der Plug-in-Schnittstelle eine ideale Testplattform für Planungsverfahren auf Ebene des VM-Schedulers.

Mit Hilfe dieser Plug-in-Schnittstelle sind im Zuge der Entwicklung von Xen mehrfach verschiedene Scheduler Techniken erprobt und ausgetauscht worden. So erwähnt [BDF+03] noch den BVT-Scheduler[1], während heute standardmäßig die Auswahl zwischen einem „Credit-" und einem „SEDF-"-Scheduler besteht. Ersterer ist ein $(3, 3)$-beschränkter Scheduler zur anteiligen Verteilung von Rechenlasten auf mehrere physische Prozessoren (siehe auch 4.1.2), letzterer implementiert einen von den Autoren als „simplified EDF" bezeichneten Algorithmus[2]. Der „Credit-"Scheduler ist derzeit der allgemein akzeptierte Standard-Scheduler für Xen.

[1] „BVT" steht für *Borrowed virtual Time* – dieser Scheduler ist in [DC99] beschrieben

[2] Über diesen Scheduler findet sich keine umfassende Beschreibung. Laut [Chi07] wird der „SEDF-"Scheduler in absehbarer Zeit aus der Quellcodebasis verschwinden.

5.3.2 Nutzung der Credit-Scheduler-Infrastruktur

Die Organisation der Plug-in-Schnittstelle von Xen sieht vor, dass beim Start des Systems durch eine entsprechende Kommandozeilenoption einer der im Virtual Machine Monitor integrierten Scheduler ausgewählt wird. Dieser übernimmt daraufhin exklusiv die Aufgabe des VM-Schedulers. Der hier implementierte Scheduler ist damit insofern ungewöhnlich, als er den vorhandenen „Credit-"Scheduler von Xen nicht vollständig verdrängt. Wie in 4.4 erläutert, arbeiten alle von Nicht-Echtzeitanwendungen genutzten virtuellen Maschinen unter einem gemeinsamen Scheduler, der eine anteilige Verteilung der für Nicht-Echtzeitprozesse gemeinschaftlich verfügbaren Rechenkapazitäten auf diese virtuellen Maschinen vornimmt.

Der bereits vorhandene Credit-Scheduler ist für für diese Aufgabe gut geeignet, deshalb wird er einfach weiterverwendet. Nur die für Echtzeit-Anwendungen genutzten virtuellen Maschinen unterstehen der Kontrolle des neu implementierten, prioritätsgesteuerten Schedulers. Alle übrigen werden durch den Credit-Scheduler gesteuert. Dazu wird bei jedem Aufruf einer der Funktionen des Schedulers geprüft, ob der gerade zu behandelnde virtuelle Prozessor dem Credit-Scheduler zugeordnet ist. Falls dem so ist, wird unmittelbar zu diesem verzweigt. Der Code des Credit-Schedulers konnte nahezu unverändert beibehalten werden. Es war lediglich ein Eingriff erforderlich, um das „Accounting" zu unterbinden, falls die gerade aktuelle virtuelle Maschine nicht unter der Kontrolle des Credit-Schedulers steht: Die Accounting-Funktion hat die Aufgabe, laufend über die von den einzelnen virtuellen Maschinen genutzten Mengen an Rechenzeit Buch zu führen. Dazu wird sie zyklisch in festen Zeitabständen aufgerufen. Durch die vorgenommene Modifikation prüft die Accounting-Funktion, ob der gerade laufende virtuelle Prozessor dem Credit-Scheduler untersteht. Ist dies nicht der Fall, unterbleibt das Accounting für diesen Zyklus.

Nachteilig bei dieser „Huckepack"-Implementierung ist, dass virtuelle Maschinen bei ihrer Erzeugung fest einem der beiden Scheduler zugeordnet werden müssen. Ein späterer Wechsel zu einem anderen Scheduler ist ausgeschlossen. Grundsätzlich wäre es möglich, diese Einschränkung zu beseitigen, jedoch wären dazu größere Eingriffe in den Code des bestehenden Credit-Schedulers nötig gewesen. Da ein dynamischer Wechsel des Schedulers für die Ziele dieser Arbeit nicht erforderlich ist, wurde hiervon jedoch abgesehen.

5.3.3 Globale Periodendauer

Der wesentliche Unterschied zwischen dem in 5.2 skizzierten VM-Scheduler und dem experimentell implementierten Scheduler ist die Verwendung einer allen vir-

tuellen Prozessoren gemeinsamen, festen Periodendauer: Während in 5.2 jeder virtuelle Prozessor über ein eigenes, jeweils durch eine Ausführungszeit Δe_{sv} und eine Periodendauer Δp_{sv} konfiguriertes Zeitfenster verfügt, ist hier zwar die Ausführungszeit Δe_{sv} ebenfalls individuell, aber die Periode ist ein global einzustellender Parameter, der für alle virtuellen Prozessoren gemeinsam gilt. Diese Periodendauer wird im Folgenden mit Δp_{vm} bezeichnet.

5.3.4 Prioritätssteuerung

Ebenso wie in 4.1.5 beschrieben wird jedem virtuellen Prozessor eine individuelle, statische Priorität zugewiesen. Damit kann nach dem RMS-Verfahren geplant werden. Für die Wahl der globalen Periodendauer Δp_{vm} ist dabei zu beachten, dass Gleichung (2.5) für *alle* nach dem RMS-Verfahren geplanten virtuellen Prozessoren gelten muss, d.h. die kleinste Periodendauer aller virtuellen Prozessoren gibt die zu wählende globale Periodendauer Δp_{vm} vor.

Listing 5.1 zeigt einen Ausschnitt aus der Datenstruktur, die in dem neuen Scheduler zur Repräsentierung physischer Prozessoren verwendet wird. (Der Name des Schedulers, „PraTSched" steht für „Priority and Time driven Scheduler".)

```
/*
 * Physical CPU
 */
struct pratsched_pcpu {
    .....
    /* active background VCPUs   */
    struct pratsched_vcpu   *backgrnd[PRATSCHED_NUM_PRIO];
    /* mask of prios in use by VCPUs on this PCPU */
    priomask_t              busy_prios;
    /* mask of active background priorities */
    priomask_t              active_prios;
    /* highest active background priority */
    uint32_t                maxprio;
    ....
    /* total allocated time on this CPU */
    s_time_t                allocated_time;
    .....
};
```

Listing 5.1: *Physische Prozessoren in Xen*

Da es endlich viele Prioritätsstufen (hier: `PRATSCHED_NUM_PRIO` = 256) gibt, und da jeder Prioritätswert nur von maximal einem virtuellen Prozessor verwendet

werden kann, ist die Prioritäts-Warteschlange einfach als Vektor (`backgrnd[]` in Listing 5.1) organisiert, der durch die Priorität indiziert wird. Zum schnellen Auffinden des höchstprioren virtuellen Prozessors gibt es zusätzlich eine Prioritäts-Bitmaske (`active_prios`), in der für jede Prioritätsstufe, auf der ein rechenwilliger virtueller Prozessor existiert, ein Bit gesetzt wird. Diese Bitmaske kann mit Hilfe von optimierten Zugriffsmethoden innerhalb weniger Maschinenzyklen durchsucht werden. Eine solche Suche wird nach jeder Änderung der Bitmaske, d.h. nach dem Hinzukommen oder dem Wegfall eines virtuellen Prozessors durchgeführt, und das Ergebnis, d.h. die höchste Prioritätsstufe, auf der ein rechenwilliger virtueller Prozessor existiert, wird in der Variable `maxprio` abgespeichert. Der Rechenaufwand zum Finden des höchstprioren virtuellen Prozessors ist damit, ebenso wie der Aufwand zum Ein- und Austragen von virtuellen Prozessoren in die, bzw. aus der Warteschlange annähernd konstant und insbesondere unabhängig von der Anzahl rechenwilliger virtueller Prozessoren[1].

Zur Buchführung über die auf einem physischen Prozessor bereits verwendeten Prioritätswerte wird eine weitere Bitmaske (`busy_prios`) verwendet. Ist in dieser Maske ein Bit gesetzt, so kann kein weiterer virtueller Prozessor mit derselben Priorität erzeugt werden.

5.3.5 Zeitfenster

Wie bei dem in 5.2 skizzierten VM-Scheduler ist der Zeitkonsum eines virtuellen Prozessors pro Periode beschränkt, womit verhindert werden kann, dass einzelne hoch-priorisierte virtuelle Prozessoren ihren physischen Prozessor monopolisieren können. Auf diese Weise wird die notwendige Fehlerlokalität für Hintergrundaktivitäten sichergestellt (vgl. 4.1.5).

Da dieser Scheduler mit einer festen Periodendauer arbeitet, wird hier zum Erfassen der Auslastung ein Zähler der insgesamt auf diesem Prozessor vergebenen Zeitanteile an der gesamten Periodendauer verwendet. Dieser Zähler (`allocated_time` in Listing 5.1) wird initial auf einen konfigurierbaren Startwert gesetzt, womit der Mindestfortschritt für Nicht-Echtzeitsysteme garantiert werden kann.

5.3.6 Virtuelle Prozessoren

Virtuelle Prozessoren werden in dem neuen Scheduler wie in Listing 5.2 gezeigt repräsentiert.

[1] D.h., es handelt sich hier um einen „O(1)-Scheduler".

```
struct pratsched_vcpu {
    ...
    struct vcpu *vcpu;
    ...
    /* state: either runnable or stopped                    */
    enum { RUN, STOP} state;
    /* priority (0 .. PRATSCHED_NUM_PRIO)                   */
    uint32_t prio;
    /* time VCPU was last started                           */
    s_time_t started_at;
    /* time remaining in current major time frame           */
    s_time_t time_remaining;
    ...
    /* total time allocated within major time frame         */
    s_time_t time_per_mtf;
    /* sequence number of major cycle when
       time_remaining was last replenished                  */
    uint32_t last_replenish;
    /* stopped because all time was used
       i.e.: awaiting replenish                             */
    int      await_replenish;
    struct list_head replenish_q_elem;
    ...
};
```

Listing 5.2: *Virtuelle Prozessoren in Xen*

Die bereits eingeführte Affinitätsmenge wird hier zweckmäßig als Bitmaske, die *Affinitätsmaske* geführt. Diese taucht *nicht* in Listing 5.2 auf, da sie bereits von der generischen Schicht in Xen bereitgestellt wird. Das Struktelement `vcpu` ist ein Zeiger auf eine von Xen verwaltete Datenstruktur, die insbesondere auch diese Bitmaske enthält.

Neben den selbsterklärenden Feldern `state` und `prio` zeigt Listing 5.2 auch die Zeitkoordinate `started_at` in der jeweils der letzte Startzeitpunkt des virtuellen Prozessors gespeichert wird, und den Zähler `time_remaining`, der die in der laufenden Periode durch diesen virtuellen Prozessor noch zu beanspruchende Rechenzeit angibt. Dieser Zähler müsste prinzipiell zu Beginn jeder neuen Periode bei allen virtuellen Prozessoren auf den Anfangswert `time_per_mtf` zurückgesetzt werden. Dies würde allerdings dazu führen, dass der Scheduler zu jedem Periodenbeginn die Datenstrukturen aller virtuellen Prozessoren manipulieren müsste. Um hier den Aufwand zu verringern, gibt es eine „Replenish-Queue”,

d.h. eine Warteliste, in der virtuelle Prozessoren eingereiht werden falls sie durch das Ende ihres Zeitfensters unterbrochen wurden. Zu Beginn der neuen Periode werden dann nur die in der „Replenish-Queue" vorgefundenen virtuellen Prozessoren aktualisiert. Für Prozessoren, die ihre Zeitscheibe nicht aufbrauchen, wird in `last_replenish` die Sequenznummer der letzten Periode gespeichert, während der zum letzten Mal der Zeitzähler aktualisiert wurde. Wird der virtuelle Prozessor in einer neuen Periode erstmalig gestartet, so lädt der Scheduler an dieser Stelle den Zeitzähler neu. Ähnlich wie bereits bei der Organisation Prioritäts-Warteschlange wird auch hier durch diese Optimierungsmaßnahmen vermieden, dass die Laufzeit des Schedulers vom Systemzustand (z.B. der Anzahl rechenwilliger Prozesse) abhängig wird.

Kapitel 6

Bewertung

Die in Kapitel 5 beschriebene Scheduler-Infrastruktur für Virtual Machine Monitore soll nun mit Hilfe praktischer Messungen bewertet werden. Ziel der Infrastruktur ist die sichere Koexistenz von Nicht-Echtzeit- und Echtzeitgastsystemen in einem gemeinsamen Rechensystem. Zur ihrer Bewertung ist daher zu untersuchen, in welchem Umfang dieses Ziel erreicht wird. Einerseits müssen Echtzeitsysteme in der Lage sein, ohne die Kenntnis der übrigen im System existierenden virtuellen Maschinen Zeitgarantien gegenüber ihren Anwendungen zu geben. Andererseits sollen die von den Echtzeitsystemen nicht genutzten Rechenzeiten den Nicht-Echtzeitsystemen zufallen. Darüber hinaus sollten die einhaltbaren Fristen so kurz sein, dass damit harte Echtzeitaufgaben mit Reaktionszeiten im Submillisekundenbereich bearbeitet werden können.

In Abschnitt 6.2 werden entsprechende Experimente beschrieben und die erhaltenen Messergebnisse diskutiert. Zuvor wird jedoch kurz das eingesetzte Konzept zur Messung relativer Rechenleistungen erläutert.

6.1 Messen von „Rechenleistung”

Zur Messung von Rechenleistungen wird hier der „Dhrystone”-Benchmark [Wei84]) verwendet. Dieser führt in einer Schleife eine Reihe von Operationen auf einer begrenzten Anzahl von speicherresidenten Datenobjekten (Strings, Integer-Variablen, etc.) aus. Er testet damit im Wesentlichen die Leistung der Integer-Arithmetik und der Speicherschnittstelle des Prozessors. Die Schleife wird so lange wiederholt durchlaufen, bis ein hinreichend genau messbares Zeitintervall verstrichen ist. Die Anzahl der Schleifendurchläufe pro Sekunde wird dann als

„Dhrystone-MIPS"-Wert zurückgeliefert. Die auf diese Weise gemessenen Leistungswerte hängen offensichtlich neben der eigentlichen Leistungsfähigkeit des Prozessors auch von den Optimierungsfähigkeiten des verwendeten Compilers, von der Geschwindigkeit des Speicherzugriffs und von der Größe und Effizienz der Caches ab. Daher sind diese Werte nur begrenzt zur Beurteilung der absoluten Rechenleistung einer Maschine tauglich.

Diese Einschränkung gilt mehr oder weniger stark ausgeprägt wohl für alle jemals eingesetzten Benchmark-Programme. Die von einem Benutzer empfundene Rechenleistung hängt neben der Arbeitsgeschwindigkeit des Prozessors noch von einer Vielzahl weiterer Faktoren und unter Anderem auch von der betrachteten Anwendung selbst ab. So gesehen dürfte bereits der Anspruch, die Leistung eines Rechensystems anwendungsneutral beziffern zu wollen, unerfüllbar bleiben. Ein Kritikpunkt speziell am Dhrystone-Benchmark ist dessen große Cache-Lokalität: Die Gesamtmenge der Datenobjekte, auf denen der Benchmark arbeitet, ist so klein, dass sie bei moderneren Prozessoren meist vollständig im Cache Platz finden. Somit geht die Geschwindigkeit der Speicherschnittstelle nicht mehr in das Benchmark-Ergebnis ein, da alle Operanden in Cache liegen. Bei den meisten realen Anwendungen trifft dies nicht im gleichen Maße zu, weshalb die durch den Dhrystone-Benchmark prognostizierte Leistung unter realen Anwendungsbedingungen in der Regel nicht erreicht wird.

Für die hier durchzuführenden Messungen sind jedoch absolute Werte der Rechenleistung ohnehin uninteressant: Es werden relative Leistungsverteilungen sowie relative Leistungseinbußen, verursacht durch Prozesswechsel, ermittelt. Die gemessenen Leistungen werden dazu stets in Prozent eines Maximalwertes angegeben, der die Leistung des Rechensystems für den Fall angibt, dass ein einziger Benchmark-Prozess ununterbrochen auf einem Prozessor arbeitet. Dieser Maximalwert wird für jede getestete Plattform in einer anfänglichen Kalibrierung ermittelt. Dazu wird eine virtuelle Maschine so konfiguriert, dass sie während der gesamten Dauer der Kalibrierung die alleinige Kontrolle über ihren Prozessor erhält: Ihr Zeitfenster wird so groß und ihre Priorität so hoch gewählt, dass ein vollständiger Durchlauf durch den Dhrystone-Benchmark ohne Unterbrechungen garantiert ist. Der so ermittelte Dhrystone-Wert entspricht somit der vollen Rechenleistung (100 %) des Systems. Für die späteren Messungen wird exakt der gleiche Code des Dhrystone-Benchmarks verwendet, sodass Unterschiede durch verschiedene Arten der Optimierung etc. ausgeschlossen sind. Alle gemessenen Leistungswerte werden stets auf den in der Kalibrierphase gemessenen Maximalwert bezogen.

6.2 Nachweise der zeitlichen Isolation

Virtual Machine Monitore bieten in der Regel eine vollständige *räumliche* Trennung der virtuellen Maschinen: Jede von ihnen existiert in einem eigenen Adressraum und hat Zugriff auf „ihre" eigenen Betriebsmittel, sodass eine gegenseitige Beeinflussung virtueller Maschinen ausgeschlossen ist. Dies gilt jedoch nicht für das Betriebsmittel „Rechenzeit": In 4.1.2 wurde am Beispiel von Xen demonstriert, dass in der Tat eine erhebliche zeitliche Abhängigkeit zwischen verschiedenen virtuellen Maschinen besteht. Dies liegt in der Natur der verwendeten anteiligen Prozessorzuteilung (und ist im Server-Umfeld durchaus gewollt).

6.2.1 Lastverteilung

Die neue Scheduler-Infrastruktur bietet die Möglichkeit, durch entsprechende Priorisierung und zyklische Zuweisung definierter Zeitvolumina die für eine bestimmte virtuelle Maschine zur Verfügung stehende Rechenzeit unabhängig vom Verhalten anderer virtueller Maschinen zusichern zu können. Um diese Eigenschaft nachzuweisen, wird folgende Anordnung verwendet:

- In einer ersten virtuellen Maschine (VM1) wird ein Dhrystone-Benchmark ausgeführt. Die Anzahl der Schleifendurchläufe wird dabei so groß gewählt, dass die Gesamtdauer eines Benchmarks um mehrere Größenordnungen über der Zykluszeit des VM-Schedulers liegt. Der vom Benchmark gelieferte Wert stellt somit in guter Näherung den zeitlichen Mittelwert der für die betreffende virtuelle Maschine verfügbaren Rechenleistung dar.
- Die virtuelle Maschine VM1 arbeitet zunächst alleine [1], und es wird die von ihr konsumierte Rechenleistung ermittelt. Anschließend wird eine zweite virtuellen Maschine (VM2) angelegt, in der der gleiche Benchmark als Anwendung läuft, sodass auch VM2 in Konkurrenz zu VM1 sämtliche ihr zur Verfügung stehende Rechenzeit in Anspruch nimmt.

Für eine erste Messung werden beide virtuellen Maschinen unter dem Standard-Scheduler von Xen (dem „Credit-"Scheduler) ausgeführt. Da dieser Scheduler eine anteilige Zuteilung anstrebt, sollte die durchschnittliche Rechenleistung von VM1 stark davon abhängen, ob auch VM2 arbeitet: Arbeitet nur VM1, so sollte

[1] abgesehen von dem Linux-System in Domain0, das zum Betrieb von Xen erforderlich ist. Dieses wird jedoch während der Messungen nicht angesprochen, d.h. es konsumiert annähernd keine Rechenleistung

Maschine	Anz. VM	Leistung VM1	Leistung VM2
Intel Celeron@400MHz	1	100.2%	-
Intel Celeron@400MHz	2	50.7%	48.8 %
AMD Turion @1.6GHz	1	100 %	-
AMD Turion @1.6GHz	2	48.8 %	50.6%

Tabelle 6.1: Leistungsverteilung durch den Xen Credit Scheduler.

sie annähernd die gesamte verfügbare Rechenleistung erhalten, arbeiten hingegen beide virtuelle Maschinen, so sollte jede von ihnen in etwa die Hälfte der Rechenleistung erhalten.

Tabelle 6.1 zeigt die auf verschiedenen IA-32 Plattformen gemessenen Leistungsverteilungen durch den Credit-Scheduler. Im Rahmen der Messgenauigkeit entspricht sie den erwarteten Werten.

Bei einer zweiten Messung wird nun VM1 unter dem prioritätsgesteuerten Scheduler ausgeführt. Ihr wird ein periodisches Zeitfenster zugeteilt, dessen Länge Δe_{sv} einen variablen Anteil der gesamten Periodendauer Δp_{sv} beträgt. VM2 arbeitet wie zuvor unter dem „Credit-"Scheduler. Bei diesem Experiment sollte die durchschnittliche Rechenleistung von VM1 proportional zum Anteil ihres Zeitfensters an der gesamten Zykluszeit sein, und sie sollte insbesondere unabhängig davon sein, ob VM2 gleichzeitig aktiv ist, oder nicht.

Abbildung 6.1 zeigt die gemessene Verteilung der Rechenleistung auf die virtuellen Maschinen VM1 und VM2 in Abhängigkeit vom prozentualen Dauer des Zeitfensters von VM1 bezogen auf die Periodendauer. Es wurden Messungen mit Periodendauern Δp_{sv} von einer, zehn und hundert Millisekunden durchgeführt, bei denen das Zeitfenster von VM1 von 10% bis 90% der Zykluszeit, also von 100 μs bis 900 μs, 1 ms bis 9 ms oder 10 ms bis 90 ms variiert wurde. Dabei wurden jeweils die Rechenleistungen von VM1 und –falls existent– von VM2 gemessen. Es zeigt sich das erwartete Verhalten: Die VM1 zugeteilte Rechenleistung ist proportional zum Anteil der Ausführungszeit an der Periode, und sie ist unabhängig davon, ob VM2 auf niedrigerer Priorität gleichzeitig aktiv ist (siehe die mit „nur VM1", bzw. mit „beide, VM1" bezeichneten Graphen). Falls VM2 existiert, so nimmt die ihr zugeteilte Rechenleistung im gleichen Maße ab, wie die von VM1 zunimmt, d.h. sie erhält die verbleibende, von VM1 nicht genutzte Rechenleistung (siehe die mit „beide, VM2" bezeichneten Graphen). Bei Vernachlässigung der umschaltungsbedingten Verluste sollte demnach die Summe der Rechenleistungen der beiden virtuellen Maschinen stets 100% betragen. Abbildung 6.1 zeigt, dass diese Summe tatsächlich zwischen 95% und 100% liegt (siehe die mit „VM1+VM2" bezeichneten Graphen).

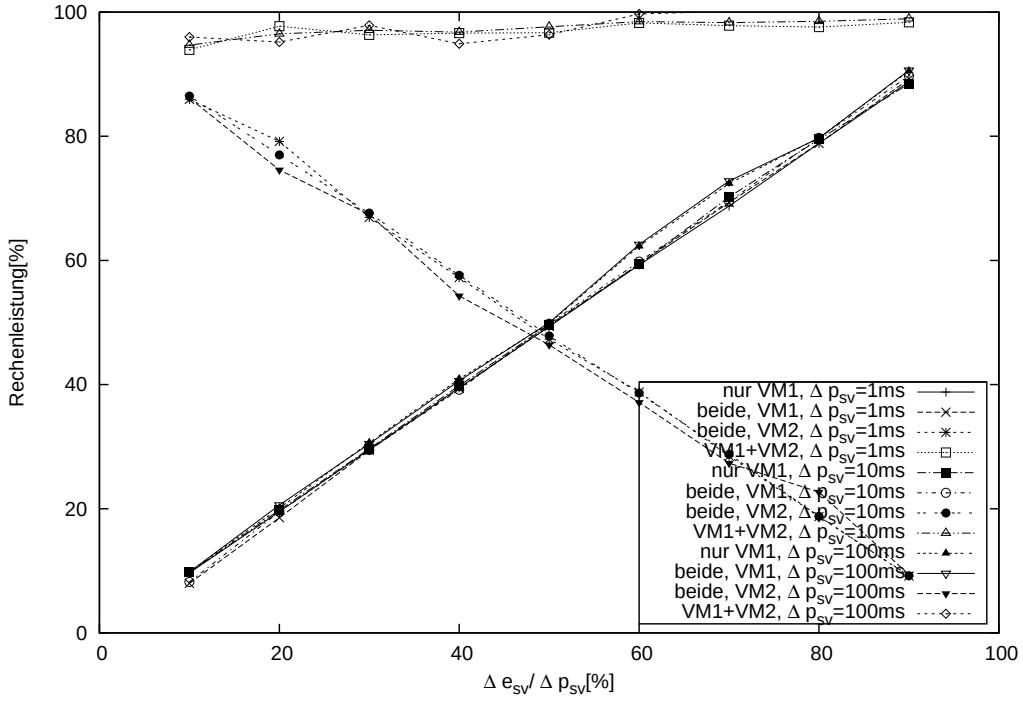

Abbildung 6.1: Leistungsverteilung unter dem neuem Scheduler.

6.2.2 Latenzzeiten

Sie zeitliche Abhängigkeit zwischen verschiedenen virtuellen Maschinen, die in 4.1.2 festgestellt wurde, äußerte sich nicht nur in einer Variabilität der zugeteilten Rechenleistung, sondern auch in starken Schwankungen der beobachteten Latenzzeiten. Nachdem das vorangegangene Experiment Rechenleistungen betrachtet hat, werden in diesem Experiment nun Latenzzeiten gemessen.

Die Anordnung ist dabei ähnlich: Eine virtuelle Maschine VM1 führt ein Benchmark-Programm aus, während eine zweite virtuelle Maschine VM2 wechselweise aktiv ist und das Dhrystone-Programm ausführt, oder nicht existiert. VM2 arbeitet –falls existent– unter dem Credit-Scheduler, VM1 arbeitet wahlweise unter dem Credit-Scheduler oder unter dem prioritätsgesteuerten Scheduler. Anders als zuvor führt VM1 hier allerdings nicht das Dhrystone-Programm aus, sondern stattdessen den gleichen Benchmark, der schon in 4.1.2 verwendet wurde, um Latenzzeiten zu erfassen.

Da das Experiment in 4.1.2 mit einem Einprozessorrechner älterer Bauart (Pentium II @800MHz) durchgeführt wurde, wird es hier auf einem aktuelleren Rechner mit Multicore-Prozessor (AMD Turion @1.6GHz) nochmals wiederholt, um

vergleichbare Ergebnisse zu erhalten.

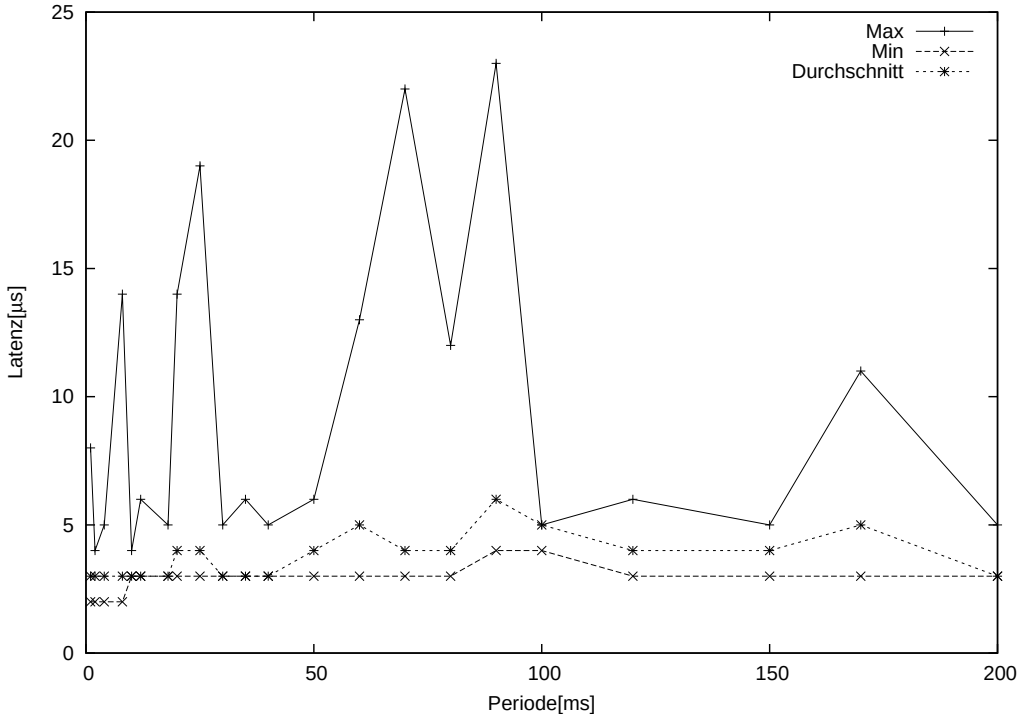

Abbildung 6.2: Latenzzeit eines periodischen Prozesses unter dem Credit-Scheduler: VM2 nicht vorhanden.

Dabei zeigt sich zunächst überraschend, dass bei dem Experiment ohne Last durch VM2 (s. Abbildung 6.2) die Latenzzeiten mit $\leq 23\mu s$ durchaus niedrig sind, während sie bei dem früheren Versuch (siehe 4.1.2) bis zu $10ms$ betrugen. Der Grund für diesen Unterschied konnte durch ein Studium des Xen-Quellcodes identifiziert werden: Der bei 4.1.2 verwendete Rechner besaß keinen „APIC-Timer". Hierbei handelt es sich um einen Uhrenbaustein, der erst in neueren PC-Rechnern (insbesondere solchen mit mehreren Prozessoren) eingebaut wird. Ersatzweise verwendete Xen den „PIC-Timer", der in allen PC-Rechnern vorhanden ist. Dieser kann im Gegensatz zum „APIC-Timer" aber nur Interrupts mit einer festen Frequenz liefern. Xen programmiert den „PIC-Timer" auf eine Frequenz von 100Hz, sodass er Interrupts im Abstand von $10ms$ erzeugt. Dieses Zeitintervall bestimmt die maximale Auflösung der Software-Timer von Xen, was die hohen Latenzzeiten im unbelasteten Fall erklärt.

Im belasteten Fall, d.h. mit aktiver VM2 steigen die gemessenen Latenzzeiten auf bis zu $60ms$ an, was gegenüber den niedrigen Werten ohne Last einen Sprung um mehrere Zehnerpotenzen bedeutet (siehe Abbildung 6.3). Hier zeigt sich wieder

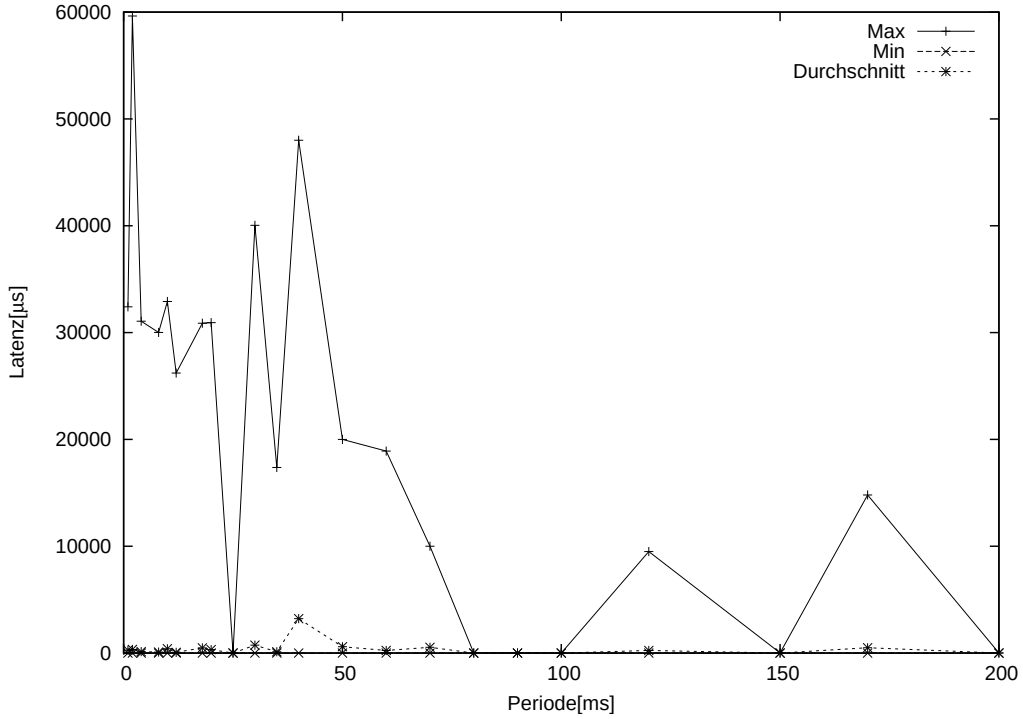

Abbildung 6.3: Latenzzeit eines periodischen Prozesses unter dem Credit-Scheduler: VM2 unter Vollast.

das gleiche Bild wie in 4.1.2, d.h. gleichgültig, ob Xen mit dem grobgranularen „PIC-Timer" oder dem wesentlich genaueren „APIC-Timer" arbeitet, hängt die in einer virtuellen Maschine beobachtete Latenzzeit in hohem Maß von der Aktivität anderer virtueller Maschinen ab. Eine zeitliche Isolation ist nicht gegeben.

Wird bei ansonsten gleichen Bedingungen VM1 unter dem prioritätsgesteuerten Scheduler ausgeführt, so ergibt sich wiederum ein zunächst überraschendes Bild (siehe Abbildungen 6.4 und 6.5): Offensichtlich wurden –wenn auch selten– *negative* Latenzzeiten gemessen. Dies bedeutet, dass der Benchmarkprozess *vor* seiner Bereitzeit aktiviert wurde. Der Grund für dieses unerwartete Verhalten fand sich abermals nach eingehendem Studium des Xen-Quellcodes:

Der Benchmark-Prozess blockiert unter Verwendung der Timer-Funktion des Betriebssystems für eine programmierte Zeit, d.h. die Bereitzeit des Prozesses wird durch den Ablauf dieses Timers festgelegt. Als Betriebssystem wurde für diesen Test das Gastsystem „mini-OS" benutzt, das standardmäßig in der Xen-Auslieferung enthalten ist. Dieses Gastsystem bildet seine Timer-Funktion unmittelbar auf die Timer-Funktion des Virtual Machine Monitors ab. Zur Implementierung dieser Funktion verwendet Xen den „APIC-Timer" des Rechners. Die-

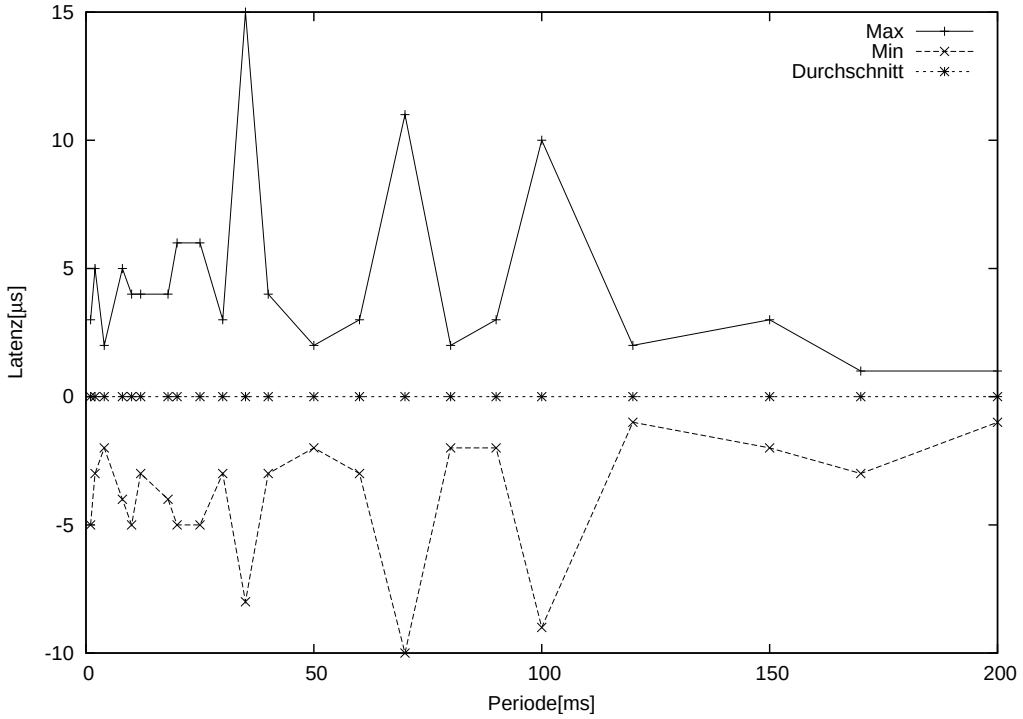

Abbildung 6.4: Latenzzeit eines periodischen Prozesses unter dem prioritätsgesteuerten Scheduler: VM2 nicht vorhanden.

ser wird jeweils so programmiert, dass er bei Ablauf des nächsten anstehenden Software-Timers einen Interrupt auslöst. Der Interrupt-Handler weckt dann den Prozess, der auf den gerade abgelaufenen Software-Timer blockiert war und programmiert ggf. den „APIC-Timer" neu. An dieser Stelle nimmt Xen eine Optimierung vor: Wenn mehrere Prozesse Software-Timer gesetzt haben, deren Ablaufzeiten hinreichend nahe[1] beieinander liegen, so fasst Xen diese zu einem einzigen Timer zusammen und weckt alle zugehörigen Prozesse bei nur einem Interrupt auf. Auf diese Weise werden viele, eng beieinanderliegende Timer-Interrupts vermieden, doch wird damit in Kauf genommen, dass Software-Timer an Genauigkeit verlieren und, wie hier beobachtet, in einigen Fällen auch zu früh ablaufen können.

Von dieser hiermit erklärten Eigenheit abgesehen zeigt jedoch ein Vergleich der Abbildungen 6.4 und 6.5 deutlich, dass die durch die virtuelle Maschine VM2 eingebrachte Last nahezu keinen Einfluss mehr auf das Verhalten von VM1 hat. Im Gegensatz zum Credit-Scheduler, bei dem die Latenzzeiten beim Hinzufügen

[1] Der Parameter, der diese Entscheidungsschwelle definiert, heißt bezeichnenderweise „TIMER_SLOP".

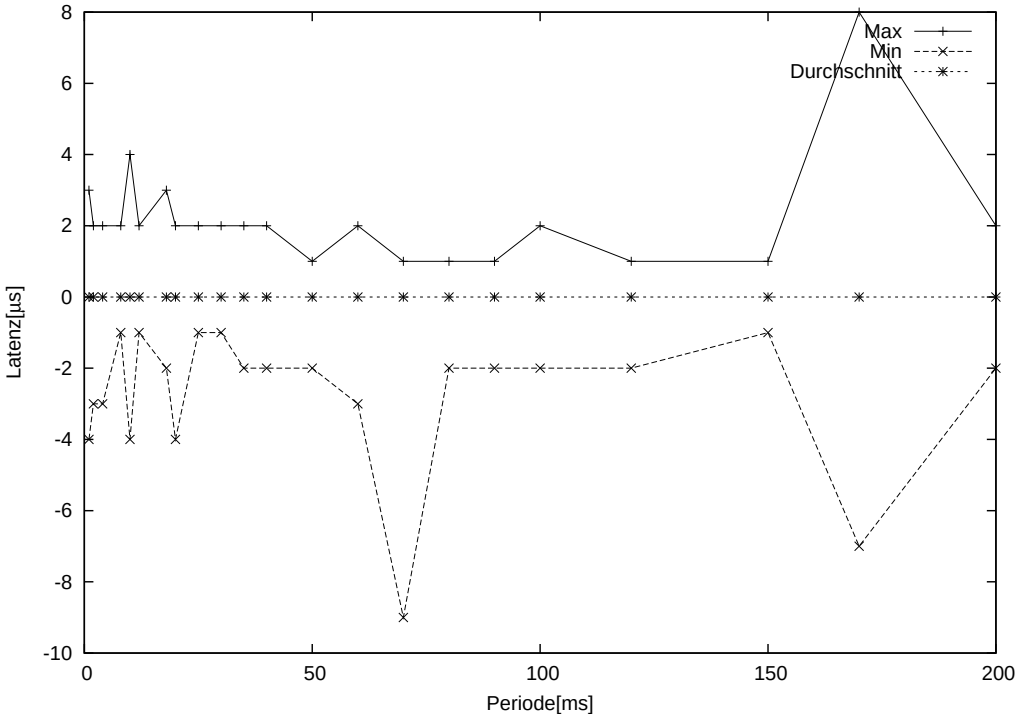

Abbildung 6.5: Latenzzeit eines periodischen Prozesses unter dem prioritätsgesteuerten Scheduler: VM2 unter Vollast.

dieser Last um mehrere Zehnerpotenzen anstiegen, scheinen sie hier sogar geringfügig zu sinken. In allen beobachteten Fällen ist die Latenzzeit unabhängig von der Last durch VM2 auf einen Bereich von ca. $\pm 15 \mu s$ beschränkt liegt damit in der gleichen Größenordnung, wie die Latenzzeiten moderner Echtzeitbetriebssysteme ohne Virtualisierungsschicht.

6.3 Fazit

Die gezeigten Messergebnisse demonstrieren, dass mit Hilfe des neuen, prioritätsgesteuerten Schedulers virtuelle Maschinen nicht nur „räumlich" sondern auch zeitlich von anderen virtuellen Maschinen entkoppelt werden können. Damit sind diese in der Lage, gegenüber ihren Prozessen Garantien über fristgerechte Ausführung abzugeben, ohne dabei das Verhalten anderer virtueller Maschinen kennen zu müssen. Zugleich ist es aber auch möglich, dass Rechenzeit, die von solchen Echtzeitsystemen nicht genutzt wird, anderen virtuellen Maschinen zufällt und von diesen zur Nicht-Echtzeitverarbeitung eingesetzt wird. Die erzielten Latenzzei-

ten sind durchaus konkurrenzfähig zu Echtzeitsystemen, die ohne Virtualisierung arbeiten.

Kapitel 7

Zusammenfassung und Ausblick

In dieser Arbeit wurden Methoden zur Erweiterung von Virtualisierungsumgebungen mit dem Ziel, Echtzeitverarbeitung innerhalb virtueller Maschinen zu ermöglichen, erarbeitet. Als Hardware-Grundlage wurden dabei insbesondere symmetrische Mehrprozessorsysteme betrachtet, deren besondere Eigenschaften nicht nur auf der Ebene der Anwendungsprogrammierung, sondern auch in Betriebssystemen und bis „hinunter" zu den im Fokus dieser Arbeit stehenden Virtual Machine Monitoren zu berücksichtigen sind.

Beiträge dieser Arbeit

Modell zur Kapselung von Prozessmengen

In Kapitel 3 wurde ein Modell zur Beschreibung von Echtzeit- und Nicht-Echtzeitanwendungen innerhalb einer Hierarchie von Prozessen eingeführt. Aufbauend auf diesem Modell der Stellvertreterprozesse wurden Methoden für die Planung unterbrechbarer periodischer Prozesse hergeleitet, die auf Gastsystemen innerhalb einer Virtualisierungsumgebung arbeiten. Solche Prozessmengen können gemeinsam mit ihren Betriebsystemen insgesamt als unterbrechbare periodische Prozesse dargestellt („gekapselt") werden. Dabei ist es mit Hilfe der gezeigten Methoden möglich, die Prozessparameter des Stellvertreterprozesses aus den Prozessparametern der darin gekapselten Prozesse zu ermitteln. Daraus ergibt sich die Möglichkeit, diese Stellvertreterprozesse auf der Ebene des Virtual Machine Monitors wiederum nach einem der bekannten Verfahren EDF oder RMS einzuplanen.

Grenze der Anwendbarkeit anteiliger Prozessorzuordnung

In einem weiteren Abschnitt von Kapitel 3 wurden die praktischen Grenzen für den Einsatz der Planungsmethode der anteiligen Prozessorzuteilung aufgezeigt: Dieses Planungsverfahren erlaubt unter Annahme infinitesimaler Einheiten der Zeitzuteilung (sog. „Quanten") eine kontinuierliche Verteilung von Rechenressourcen. Da dies dem Modell einer Virtuellen Maschine sehr gut entspricht, werden derartige Planungsverfahren heute in nahezu allen verbreiteten Virtualisierungsumgebungen angewendet. Wie sich aber in der Praxis zeigt, sind die idealisierten Bedingungen, von denen diese Verfahren ausgehen, im Zusammenhang mit Echtzeitanwendungen nicht immer gegeben. Dass eine untere Grenze für die Größe der Quanten existiert, ist intuitiv klar, doch ist es schwierig, diese Grenze zu beziffern. Um hier zu einer praktikablen Abschätzung zu gelangen, wurde ein neues Modell zur Beschreibung der Rechenzeitverluste eingeführt, die im Zuge von Prozessumschaltungen entstehen. Dieses Modell wurde sodann durch praktische Messungen „kalibriert". Diese Kalibrierung kann für beliebige Prozessoren durchgeführt werden, d.h. die Methode ist hardwareunabhängig. Damit steht nun ein Werkzeug zur rechnerischen Abschätzung der zu erwartenden Umschaltverluste in einem Rechensystem zur Verfügung.

Ein Simulator für verschiedene Verfahren zur anteiligen Prozessorzuteilung wurde geschaffen, der die so berechneten Umschaltverluste bei den erzeugten Zeitplänen mit einkalkuliert. Auf diese Weise konnten die bei der Wahl eines bestimmten Quantums zu erwartenden Umschaltverluste mit den zu erwartenden Latenzzeiten von Echtzeitprozessen in Beziehung gesetzt werden. Dabei zeigte sich, dass Latenzzeiten im Submillisekundenbereich, wie sie von einigen Echtzeitanwendungen gefordert werden, nur durch sehr kurze Quanten und damit unter Inkaufnahme extrem hoher Verluste an Rechenleistung erreichbar sind. Verfahren zur anteiligen Prozessorzuteilung sind damit nur für einen Teil der auf ein eingebettetes System zukommenden Aufgaben realistisch anwendbar.

Diese Erkenntnis lieferte die Motivation zu einer Klassifikation der anstehenden Rechenlasten (s.u.): Je nach der Härte ihrer Zeitanforderungen und nach der Größenordnung der einzuhaltenden Fristen können Echtzeitanwendungen entweder nach der anteiligen Prozessorzuteilung geplant werden, oder es müssen andere Planungsverfahren für sie gefunden werden.

Scheduler-Infrastruktur zur Koexistenz von Anwendungen mit unterschiedlichen Zeitanforderungen in einer Virtualisierungsumgebung

In Kapitel 4 wurden zunächst verschiedene Möglichkeiten der Koexistenz von Echtzeit- und Nicht-Echtzeitanwendungen in einem Rechensystem betrachtet.

Diese Koexistenz ist erforderlich, da eine Virtualisierungsumgebung als generische Plattform mit unterschiedlichsten Zeitanforderungen von Seiten der Gastsysteme konfrontiert sein kann. Sie ist zugleich auch wünschenswert, da so Betriebsmittel, die in einem „reinen" Echtzeitsystem infolge der „worst-case" Planung ungenutzt bleiben würden, von Nicht-Echtzeitanwendungen genutzt werden können. Nach der Analyse einiger (zum Teil nicht auf Virtualisierung beruhender) Ansätze wurden am Beispiel der Virtualisierungsumgebung Xen zusätzlich experimentelle Untersuchungen vorgenommen. Dabei zeigte sich, dass bei der bestehenden Form von Xen Echtzeitverarbeitung in virtuellen Maschinen immerhin eingeschränkt möglich ist, dass aber insbesondere keine zeitliche Entkopplung zwischen virtuellen Maschinen besteht. Dieses experimentelle Resultat konnte auch anhand einer einfachen analytischen Betrachtung bestätigt werden.

Ausgehend von einer Klassifikation der Rechenlasten anhand ihrer Zeitanforderungen konnte anschließend die Struktur eines Schedulers für virtuelle Maschinen (zunächst nur für Einprozessorsysteme) angegeben werden, die die speziellen Anforderungen von Echtzeit- und Nicht-Echtzeitsystemen gleichermaßen berücksichtigt. Diese basierte auf einer Kombination aus anteiliger Prozessorzuordnung und Prioritätssteuerung: Erstere erlaubt eine gleichmäßige Rechenlastverteilung zwischen Anwendungen ohne strenge Zeitanforderungen, letztere erlaubt unter Verwendung des Modells zur Kapselung von Prozessmengen (s.o.) exakte Planbarkeitstests und somit verlässliche Zeitgarantien für Echtzeitanwendungen.

Erweiterung der Infrastruktur für Mehrprozessorsysteme

Im letzten Abschnitt von Kapitel 3 wurde eine Unterscheidung zwischen Ein- und Mehrprozessorbetriebssystemen vorgenommen, und es wurde gezeigt, dass Mehrprozessorsysteme Annahmen über die echt-parallele Ausführung auf mehreren Prozessoren machen, die in einer Virtualisierungsumgebung nicht immer zutreffen. Unter Bezugnahme hierauf wurde im zweiten Abschnitt von Kapitel 4 die Betrachtung auf Mehrprozessorsysteme ausgedehnt. Dabei wurde eine auf [CFH$^+$03] basierende Klassifikation von Schedulern anhand der Freiheitsgrade angewendet, die diese hinsichtlich der Möglichkeiten zur Prozessmigration und der dynamischen Vergabe von Prioritäten besitzen. Aus insgesamt neun Möglichkeiten konnten zwei Klassen von Schedulern ausgewählt werden, die für Echtzeit- bzw. Nicht-Echtzeitaufgaben eine vorteilhafte Wahl darstellen: Für die Echtzeitaufgaben sind dies die so genannten $(1, 1)$- oder $(2, 1)$-beschränkten Scheduler, die keine Prozessmigration zulassen, und die statische ($(1, 1)$) oder eingeschränkt dynamische ($(2, 1)$) Prioritäten unterstützen. Für die Nicht-Echtzeitprozesse sind es die in beiderlei Hinsicht uneingeschränkten $(3, 3)$-beschränkten Scheduler. Diese Scheduler können als Erweiterungen der bereits für den Einprozessorfall iden-

tifizierten Scheduler (RMS oder EDF für Echtzeitaufgaben, anteilige Zuteilung für Nicht-Echtzeitaufgaben) angesehen werden.

Zusammenfassend wurden am Ende von Kapitel 4 die bestehenden Anforderungen an eine Scheduler-Infrastruktur für Virtual Machine Monitore in Mehrprozessorsystemen angeführt. Eine für Mehrprozessorsysteme entsprechend erweiterte Form der Scheduler-Infrastruktur, die diesen Anforderungen gerecht wird, wurde skizziert.

Prototypische Implementierung und Evaluierung der Infrastruktur für Mehrprozessorsysteme

In Kapitel 5 wurde die Implementierung der am Ende von Kapitel 4 skizzierten Scheduler-Infrastruktur beschrieben. Dabei wurden die benötigten Methoden und Datenobjekte zunächst ohne Bezug auf eine Integration in einen bestimmten Virtual Machine Monitor vorgestellt und ihre Arbeitsweise erläutert. Anschließend wurde die Integration des Schedulers in den Virtual Machine Monitor Xen beschrieben. Diese Implementierung nutzt den in Xen bereits vorhandenen „Credit"-Scheduler für den in Kapitel 4 beschriebenen Funktionsbaustein des hinsichtlich Migration wie auch dynamischer Prioritätszuordnung unbeschränkten, so genannten „fairen" Schedulers. Ein weiterer, migrationsbeschränkter Scheduler mit statischen Prioritäten zur Steuerung der Echtzeitprozesse wurde neu geschaffen. In Kapitel 6 wurde dieser Scheduler anhand verschiedener Benchmarks evaluiert. Die in Xen implementierte Scheduler-Struktur weicht von der zuvor beschriebenen Struktur insofern ab, als sie nur eine feste Periodendauer unterstützt, während die skizzierte Struktur Zeitfenster mit individueller Periodendauer und Ausführungszeit für jeden virtuellen Prozessor vorsieht. Dies vermindert jedoch die Brauchbarkeit für die Evaluation nicht. So konnte experimentell gezeigt werden, dass die Zeitzuteilung an virtuelle Maschinen, die unter dem prioritätsgesteuerten Scheduler arbeiten, unabhängig vom Verhalten anderer virtueller Maschinen garantiert werden kann. Die dabei erzielten Latenzzeiten sind durchaus konkurrenzfähig mit denen von Echtzeitsystemen, die ohne unterlagerte Virtualisierungsschicht arbeiten. Somit können nun virtuelle Maschinen unabhängig von den übrigen virtuellen Maschinen im System Planbarkeitstests für ihre Anwendungen durchführen und ihnen ggf. eine fristgerechte Ausführung garantieren. Die räumliche Isolation der virtuellen Maschinen wurde damit um eine zeitliche Isolation ergänzt.

Ausblick

Die Grenze der Anwendbarkeit der Verfahren zur anteiligen Prozessorzuordnung konnte im Rahmen dieser Arbeit aufgezeigt werden. Dies lieferte die Motivation zur Verwendung prioritätsgesteuerter (und migrationsbeschränkter) Planungsverfahren für diejenigen virtuellen Maschinen, die Echtzeitprozesse auszuführen haben. Intuitiv ist einzusehen, dass mit diesen Verfahren kürzere Latenzzeiten bei geringeren Umschaltverlusten erzielt werden können. Auch hier wäre jedoch –analog zu den Verfahren zur anteiligen Prozessorzuordnung– eine quantitative Aussage wünschenswert. So könnte, in gleicher Weise wie in Kapitel 3 für die anteilige Zuordnung geschehen, mit Hilfe eines entsprechenden Simulators die Zeitzuteilung für Prozesse ermittelt werden, die unter einer zweistufigen Hierarchie von RMS- oder EDF-Schedulern arbeiten. Dabei könnte auch wieder das Modell zur Abschätzung der Umschaltverluste mit einbezogen werden.

Aus [But05] ist bekannt, dass das EDF-Verfahren zu weniger Prozesswechseln tendiert, als das RMS-Verfahren. Auch in der genannten Arbeit wurde zur Stützung dieser Aussage eine Simulation benutzt. Leider wurden dabei die Kosten der Umschaltverluste nicht ermittelt (es wurde lediglich deren Anzahl, bzw. Häufigkeit festgestellt). Die hier vorgeschlagene Simulation könnte dagegen Umschaltverluste mit einbeziehen, was –auch wenn es sich dabei nur um Abschätzungen handelt– zu realitätsnäheren Ergebnissen führen dürfte.

Bei der Entwicklung des Modells zur Abschätzung der Umschaltverluste wurden zwei Approximationsfunktionen zur Beschreibung des zeitlichen Verlaufs des Nutzlastfaktors bei Prozesswechseln eingeführt. Eine dieser Funktionen, die den „worst-case" beschreibt, konnte mit Hilfe einer synthetischen Rechenlast experimentell nachvollzogen werden. Sie liefert naturgemäß sehr pessimistische Abschätzungen, wie sie für die Dimensionierung von Echtzeitsystemen mit harten Zeitanforderungen benötigt werden. Um darüber hinaus auch weniger pessimistische Einschätzungen zu erhalten, die dann beispielsweise auf Systeme mit „weichen" Echtzeitanforderungen anwendbar wären, wurde eher willkürlich eine Exponentialfunktion eingeführt. Über deren tatsächliche Brauchbarkeit zur Beschreibung realen Verhaltens ist bis jetzt keine Aussage möglich. (Erste Messergebnisse mit zufällig verteilten Zugriffsadressen deuten auf eine lineare Beziehung hin [Kai08]).

Die Scheduler-Infrastruktur, die im Rahmen dieser Arbeit entwickelt wurde, bietet eine geeignete Messumgebung, mit der durchschnittliche Umschaltverluste bei realen Anwendungen ermittelt werden können: Dazu könnte einem Linux System ein festes Zeitfenster gegeben werden, und die Verarbeitungsleistung von Anwenderprogrammen wie zum Beispiel Compiler, Video- oder Audio-Encoder, etc.

könnte in Abhängigkeit von der Länge der Zeitscheibe gemessen werden. Auf diese Weise lassen sich möglicherweise bessere Abschätzungen für die Kosten von Prozesswechseln gewinnen (die dann allerdings nur für „weiche" Echtzeitanwendungen verwendbar sind).

Neben der Anwendung auf Echtzeitsysteme sind diese Erkenntnisse auch für generelle Problemstellungen im Bereich der Betriebssysteme nutzbar: Die Methode zur Berechnung der zu erwartenden Prozesswechselverluste ist, nachdem sie einmal experimentell kalibriert wurde, einfach genug, um auch von einem Online-Scheduler zur Laufzeit des Systems durchgeführt zu werden. Auf diese Weise können die Heuristiken solcher Scheduler verbessert werden. So könnte beispielsweise ein „intelligenter" Scheduler künftig dynamisch entscheiden, ob eine mögliche Migration eines Prozesses tatsächlich „lohnt", indem er die geschätzten Kosten der Migration und die auf dem Zielprozessor aktuell verfügbare Rechenkapazität gegeneinander abwiegt.

Solche Methoden zur Vermeidung ineffizienter Prozessmigrationen auf Betriebssystemebene dürften in Zukunft an Bedeutung gewinnen, da die heutigen Zwei- und Vierkern-Prozessoren nach den Ankündigungen der Prozessorhersteller erst den Anfang darstellen. In Zukunft dürfte die Fähigkeit von Software, Mehrprozessorsysteme sinnvoll zur Parallelverarbeitung zu nutzen, immer mehr zum entscheidenden Faktor für ihren Erfolg werden. Wie bereits in der Einleitung gesagt wurde, erfordert das Entwickeln von parallelen Programmen einerseits den Einsatz neuer Programmiermethoden und -Werkzeuge, andererseits aber auch die entsprechenden unterstützenden Schnittstellen von Seiten des Betriebsystems. Zur Modellierung und Implementierung solcher Schnittstellen –sei es auf der Ebene eines Betriebssystems oder der eines Virtual Machine Monitors– hat die vorliegende Arbeit einen Beitrag geliefert

Eine praktische Zielrichtung für künftige Arbeiten ist die Implementierung des beschriebenen Schedulers in einem anderen Virtual Machine Monitor: Wie dargestellt, ist Xen zwar eine gute Testumgebung zur Entwicklung von Schedulern, doch ist es für den realen Einsatz in eingebetteten Systemen denkbar ungeeignet. Nachdem der Scheduler mit Hilfe der Xen-Umgebung prototypisch entwickelt wurde, könnte er nun in einem kompakteren, für eingebettete Umgebungen geeigneten Umfeld eingesetzt werden. Es existieren mehrere Ansätze die auf Basis der Mikrokern-Technologie versuchen, Virtualisierung für eingebettete Systeme zu etablieren (z.B. „OKL4" ([Hei08]), denen aber bisher ein schlüssiges Konzept zur Unterstützung von Echtzeitanwendungen fehlt. Durch Integration der in dieser Arbeit beschriebenen Scheduler-Infrastruktur in einen solchen Mikrokern würde eine universelle Virtualisierungsplattform für eingebettete Systeme entstehen, in der Echtzeit- und Nicht-Echtzeit Anwendungen zeitlich und räumlich voneinander getrennt in einem gemeinsamen Rechner betrieben werden können.

Anhang A

`prop-share-sim`: Simulationsprogramm für anteilige Prozessorzuordnung

Das Programm `prop-share-sim` ist ein C-Programm zur Simulation verschiedener Verfahren zur anteiligen Prozessorzuteilung. Es erzeugt verschiedene Ausgabedateien, die errechnete Daten in Form von Wertepaaren enthalten, sodass sie beispielsweise mit dem OpenSource-Programm `gnuplot` grafisch ausgegeben werden können. Darüber hinaus können zusammenfassende Daten in Form einer „.csv"-Datei (comma seperated value) erzeugt werden. Solche Dateien können von den meisten Tabellenkalkulationsprogrammen eingelesen werden.

A.1 Funktion und Kommandosyntax

Das Programm wird wie hier angegeben aufgerufen:

```
Usage: ./prop-share-sim [<options>] weight0 weight1 ....
Options:
 -s <switchtime> - specify switch time (default: 10)
 -q <quantum>    - specify quantum (default: 100)
 -t <time>       - time to run (default: 200000)
 -n <name>       - name pattern for output files (default: "ps%d.%s")
 -a <alg>        - select scheduler algorithm: wrr, llf or vtrr
 -C <alg>        - select cache emulation algorithm: avg or flood
 -i              - assume one timer interrupt per process switch
 -f              - show detailed cache loading effort effect
```

```
-x          - output execution times
-e          - output net execution times
-l          - output process lags
-p          - output process schedule
-g          - output gnuplot command files
-c          - output CSV summary
-h          - add headline to CSV output
```

Listing A.1: *prop-share-sim Hilfetext*

Abhängig von den angegebenen Optionen erzeugt es eine Reihe von Dateien (s.u.). Die relativen Gewichte der Prozesse werden auf der Kommandozeile eingegeben. Die Anzahl der angegebenen Gewichte bestimmt die Anzahl der Prozesse des zu simulierenden Prozesssystems.

A.2 Optionen

Das Programm unterstützt folgende Optionen:

-s <switchtime>: Scheduler-Zeit angeben. Für jeden Aufruf des Schedulers wird diese Zeitdauer veranschlagt, d.h. keinem Prozess wird während dieses Intervalls Zeit zugeteilt. Standardwert: 10

-q <quantum>: Quantum (d.h. minimale Zeitzuteilung des Schedulers angeben. Standardwert: 100.

-t <time>: Gesamt-Laufzeit für die Simulation. Standardwert: 200000.

-n <name>: Namensmuster für die Ausgabedateien: Der String, der hier angegeben wird, muss ein „%d" und ein „%s" (in dieser Reihenfolge) enthalten. Bei den Namen der Dateien wird „%d" durch die Nummer des Prozesses ersetzt und „%s" durch die jeweilige File Extension (Details s.u.). Standardwert: „ps%d.%s".

-a <alg>: Auswahl des zu simulierenden Planungsverfahrens. Mögliche Werte sind: `wrr` = Weighted Round Robin, `llf` = Lowest Lag First oder `vtrr` = Virtual Time Round Robin.

-C <alg>: Auswahl des simulierten Cache-Füllverhaltens. Mögliche Werte sind: `avg` = Average (Exponentialfunktion) oder `flood` = Flood (Worstcase Verhalten).

-i One Shot Timer: In diesem Fall wird der Scheduler nur gerufen (bzw. die Scheduler-Zeit wird nur veranschlagt), falls der Aufruf zu einem Prozesswechsel führt, d.h. Scheduler-Aufrufe die nicht von einem Prozesswechsel gefolgt sind, unterbleiben.

-f Exakten zeitlichen Verlauf des Cache-Verhaltens anzeigen: Dieser Verlauf ist kontinuierlich (z.B. exponentiell), d.h. es müssen relativ viele Punkte berechnet werden, um diesen Verlauf exakt darzustellen. Ohne diese Option werden nur der Anfangs- und der Endpunkt berechnet. Empfehlenswert bei hoher Zeitauflösung.

-x : Erzeuge zeitlichen Verlauf der zugewiesenen Rechenzeit, Dateien „*.xtime"

-e : Erzeuge zeitlichen Verlauf der zugewiesenen Rechenzeit abzüglich Umschaltverlusten, Dateien „*.etime".

-l : Erzeuge zeitlichen Verlauf des Rückstandes („Lag"), Dateien „*.lag"

-p : Erzeuge den Ausführungszeitplan, Datei „*.sched"

-g : Erzeuge passende Kommandodateien für gnuplot. Dateien: {etime,lag,sched, xtime}.gpl

-c : Erzeuge CSV-Datei mit zusammenfassenden Informationen. Datei *.csv (wird im Append-Mode geschrieben)

-h : Erzeugen einer Kopfzeile i der .CSV-Datei.

A.3 Unterstützte Planungsverfahren

Der Simulator unterstützt drei verschiedene Planungsverfahren:

WRR : Weighted Round Robin: Hierbei wird *kein* festes Quantum verwendet, sondern den Prozessen werden Zeitscheiben zugeteilt, deren Länge proportional zu ihren Gewichten ist.

LLF : Lowest Lag First: Nach Ablauf eines Quantums wird der Scheduler aufgerufen. Er wählt jeweils den Prozess, dessen Rückstand aktuell den kleinsten Wert hat, der also aktuell den größten Zuweisungsfehler hat.

VTRR : Variante eines Verfahrens zur anteiligen Planung, das im Gegensatz zu den meisten anderen mit konstanter Scheduler-Laufzeit (d.h. O(1)) arbeitet. Siehe [NVZ01].

A.4 Erzeugte Ausgabedateien

Das Erzeugen sämtlicher möglicher Dateien muss mit entsprechenden Optionen aktiviert werden (s.o.). Einige Dateien werden für jeden einzelnen Prozess erzeugt. Diese heißen dann <*namensmuster*><*prozessnummer*>.<*dateityp*>. *Beispiel*: Die Datei `ps1.xtime` enthält den Verlauf der zugewiesenen Ausführungszeit (<*dateityp*> = textttxtime) für Prozess Nr. 1 (<*prozessnummer*> = texttt1). Es Wurde kein Namensmuster angegeben, deshalb wurde das Standardmuster „ps%d.%s" verwendet. Im Einzelnen können Folgende Dateien erzeugt werden:

<name><proz>.xtime : Enthält den zeitlichen Verlauf der dem Prozess zugewiesenen Rechenzeit als Folge von Wertepaaren (<Zeit>, <Wert>).

<name><proz>.etime : Enthält den zeitlichen Verlauf der dem Prozess zugewiesenen Rechenzeit nach Abzug der Cache-bedingten Umschaltverluste.

<name><proz>.lag : Enthält den zeitlichen Verlauf des Rückstandes („Lag") des Prozesses.

<name>.sched : Enthält den Ausführungsplan, d.h. die Nummer des aktiven Prozesses als Funktion der Zeit.

<typ>.gpl : Kommandodateien für `gnuplot`, zur bequemen Bildschirmausgabe.

<name>.csv : CSV-Datei mit Zahlreichen statistischen Werten (z.B. durchschnittliche Latenzzeit, worst-case Latenzzeit, Prozesswechselbedingte Verluste, etc.. Pro Programmlauf wird jeweils eine Zeile an die Datei angehängt, d.h. diese Datei wird stets im Append-Mode geöffnet. Wird das Programm ‚it der Option „-h" aufgerufen, so wird zusätzlich eine Kopfzeile geschrieben. Dies sollte nur bei ersten Durchlauf getan werden, damit die erzeugte Tabelle eine Kopfzeile bekommt.

A.5 Beispiel

Das Diagramm in Abbildung 2.11 lässt sich (ohne Hilfslinien) mit folgenden Kommandos erzeugen:

```
./prop-share-sim -s0 -q1 -nllf -allf -x -l -t15 4 3 3
gnuplot
gnuplot> plot "llf0.xtime" with lines, "llf0.lag" with lines
```

Listing A.2: *Beispiel 1: Erzeugen und Anzeigen einer Zeitplans*

Erläuterungen: Es werden 3 Prozesse mit den relativen Gewichten 4, 3 und 3 simuliert. Optionen:

-s0 → Scheduler-Laufzeit = 0

-q1 → Quantum = 1

-nllf → Namensmuster für Dateien = „llf%d.%s"

-allf → Wähle LLF-Scheduler

-x → Erzeuge Dateien `llf0.xtime`, `llf1.xtime`, `llf2.xtime`

-l → Erzeuge Dateien `llf0.lag`, `llf1.lag`, `llf2.lag`

-t15 Simuliere für den Zeitraum $0 \leq t \leq 15$

Das `plot`-Kommando bewirkt die grafische Ausgabe der Inhalte der Dateien `llf0.xtime` und `llf0.lag`, die durch den Programmaufruf erzeugt wurden.

Anhang B

`edf-sched`: Ermittlung brauchbarer Kombinationen $(\Delta e_{sv}, \Delta p_{sv})$ zur Kapselung von EDF-Prozessmengen

Das Programm `edf-sched` ist ein C-Programm zur iterativen Berechnung von brauchbaren Wertepaaren $(\Delta e_{sv}, \Delta p_{sv})$ zur Kapselung von nach EDF geplanten Prozessmengen (siehe 3.1.3). Es prüft für eine gegebene Prozessmenge zunächst anhand Gleichung (2.8) ob die gegebene Prozessmenge planbar ist. Ist dies der Fall, so iteriert das Programm über alle möglichen Kombinationen von Periodendauern Δp_{sv} und Ausführungszeiten Δe_{sv} und prüft anhand des in Abschnitt 3.1.3 beschriebenen Kriteriums (3.19), ob für das jeweilige Wertepaar ein brauchbarer Plan existiert. Ist dies der Fall, so wird das entsprechende Wertepaar ausgegeben. Aus allen brauchbaren Wertepaaren wird zusätzlich das optimale Wertepaar angegeben, für das sich die geringste Differenz zwischen Auslastung und Zuweisung gemäß Gleichung (3.29) ergibt. Zusätzlich kann das Programm eine Komandodatei für das Programm gnuplot erzeugen, mit dem der Zeitverlauf von Zuweisung und Bedarf für das gefundene, optimale Wertepaar grafisch dargestellt werden kann.

B.1 Funktion und Kommandosyntax

Das Programm wird wie hier angegeben aufgerufen:

```
Usage: ./edf-sched <cap1>,<per1> <cap2>,<per2> <cap3>,<per3> ....
```

Listing B.1: *edf-sched Hilfetext*

Dabei sind die `<cap>`, `<per>` jeweils Ausführungszeiten (*capacities*, Δe_i) und Periodendauern (Δp_i) der zu kapselnden, nach EDF geplanten Prozessmenge.

B.2 Optionen

Das Programm unterstützt folgende Optionen:

- -? : Obigen Hilfetext anzeigen
- -p <plotfile>: Erzeuge eine Kommandodatei für gnuplot. Falls <plotfile> nicht angegeben wird, wird eine Datei namens `edf.gnuplot` erzeugt.

B.3 Erzeugte Ausgaben

Das Programm gibt für jedes brauchbare Wertepaar eine Ausgabezeile der Form aus:

`Feasible: p_sv = <periode>, e_sv = <ausführungszeit>, Delta_U = <d`

Das optimale Wertepaar wird nochmals ausgegeben:

`Optimal: p_sv = <periode>, e_sv = <ausführungszeit>, Delta_U = <de`

Außerdem wird die Auslastung durch die Prozessmenge ausgegeben:

`U = <auslastung>`

Bei Angabe der Option `-p` wird zusätzlich eine Datei `edf.gnuplot` erzeugt.

B.4 Beispiel

Ein zu Abbildung 3.8 analoges Diagramm lässt sich (ohne Hilfslinien und Pfeile) mit folgenden Kommandos erzeugen:

```
./edf-sched -p 1,3 1.5,6 0.5,9
gnuplot
gnuplot> load "edf.gnuplot"
```

Listing B.2: *Beispiel 1: Erzeugen und Anzeigen von Zuweisung und Bedarf einer Prozessmenge*

Hinweis: In der Achsenbeschriftung des Diagramms wird die spezielle Sequenz `{/Symbol D}` zur Darstellung des griechischen „Δ” verwendet. Diese Darstellung funktioniert allerdings nur korrekt, wenn gnuplot eine PostScript-Datei ausgibt. Bei der Bildschirmausgabe erscheint die o.a. Sequenz.

Anhang C

Inhalt der CD

Auf der beiliegenden CD-ROM befinden sich folgende Verzeichnisse:

Verzeichnis	**Inhalt**
`kmod`	Linux Kernmodul zur Messung der Cache-bedingten Umschaltverluste: Quellcode, Auswerteprogramme und weitere Messdaten.
`PropShare-sim`	Simulator für Planungsverfahren zur anteiligen Prozessorzuordnung (siehe Anhang A): Quellcode und weitere Beispiele.
`Edf-feasible`	Hilfsprogramm zur Ermittlung planbarer Kombinationen von Periodendauern und Ausführungszeiten zur Kapselung nach EDF geplanter Prozessmengen (siehe Anhang B): Quellcode und Beispiele.
`Xen`	Modifikationen und Erweiterungen des Xen-Systems, insbesondere der neu entwickelte PraTD-Scheduler mit dem zugehörigen Konfigurationsprogramm `schedctrl.c`.
`tt-os`	Benchmarkumgebung für Xen, basierend auf Xen „mini-os”: Quellcode, Shell-Script zur Durchführung der in Kapitel 6 beschriebenen Tests sowie weitere Messdaten.

Weiterführende Erklärungen und Hinweise finden sich jeweils in den Dateien `liesmich.txt`. In jedem der obengenannten Verzeichnisse befindet sich eine solche Datei.

Anhang D

Lebenslauf

Dipl.-Ing. (TH) Robert Kaiser

Geburtsdatum, Ort	13. Dezember 1959, Boppard (Rhein)
Familienstand	verheiratet, zwei Kinder
Staatsangehörigkeit	Deutsch
Adresse	Riederbergstraße 23, D-65195 Wiesbaden

Schulbildung

1965-1969	Katholische Volksschule Boppard
1969-1978	Staatliches Kant-Gymnasium Boppard

Studium

1978-1981	Universität Kaiserslautern
1981-1986	RWTH Aachen

Abschlüsse

1978	Allgemeine Hochschulreife
1986	Diplom-Ingenieur (TH) Elektrotechnik

Beruflicher Werdegang

1986	Wissenschaftliche Hilfskraft am Institut für Hochfrequenztechnik der RWTH Aachen.
1987-91	Software-Ingenieur bei Eltec elektronik GmbH, Mainz. Tätigkeiten: Entwicklung von Software zur industriellen Bildverarbeitung, Portierung des Echtzeitbetriebssystems OS-9.
1992-2007	Mitbegründer der Firma Sysgo Real-Time Solutions GmbH (heute: Sysgo AG), Klein-Winternheim. Tätigkeit: Leiter Forschung und Entwicklung.
Oktober 2005-August 2007	Gastwissenschaftler im Labor für verteilte Systeme der Fachhochschule Wiesbaden.
Seit September 2007	Wissenschaftlicher Mitarbeiter im Labor für verteilte Systeme der Fachhochschule Wiesbaden.

Nebentätigkeiten

Seit 2000	Lehrbeauftragter der Fachhochschule Wiesbaden: Vorlesung und Praktikum über Echtzeitbetriebssysteme (Liste I), Vertiefungsveranstaltung über Embedded Systems, Vorlesung und Praktikum über Betriebssysteme und Rechnerarchitekturen.
Seit 2008	Lehrbeauftragter der Hochschule Furtwangen: Vorlesung über Betriebssysteme.

Mitgliedschaften

Seit 2007	GI/ITG, Fachgruppen Betriebssysteme und Echtzeitsysteme
Seit 2008	EuroSys - European Professional Society on Computer Systems

Literaturverzeichnis

[Amd67] AMDAHL, Gene M.: Validity of the single processor approach to achieving large scale computing capabilities. In: *Proceedings of AFIPS Conference* (1967), S. 483–485

[AR06] AKHTER, Shameem ; ROBERTS, Jason ; CLARK, David J. (Hrsg.): *Multicore Programming*. Richard Bowles, 2006 (Intel Press)

[ARI97] ARINC: Avionics Application Software Standard Interface / Aeronautical Radio, Inc. 1997 (ARINC Specification 653). – Forschungsbericht

[BDF+03] BARHAM, Paul ; DRAGOVIC, Boris ; FRASER, Keir ; HAND, Steven ; HARRIS, Tim ; HO, Alex ; NEUGEBAUER, Rolf ; PRATT, Ian ; WARFIELD, Andrew: *Xen and the Art of Virtualization*. 2003

[BK03] BATE, I. ; KELLY, T.: Architectural Considerations in the Certification of Modular Systems. In: *Reliability Engineering and System Safety* 81 (2003), S. 303–324

[Bol97] BOLLELLA, Gregory: *Slotted priorities: supporting real-time computing within general-purpose operating systems*, University of North Carolina at Chapel Hill, Diss., 1997. – Adviser-Kevin Jeffay

[Bom08] BOMMERT, Marc: *Synchronisation von Anwendungen in verteilten Virtualisierungsumgebungen unter Verwendung des Precision Time Protocol*, Fachhochschule Wiesbaden, FB DCSM, Diplomarbeit, Februar 2008

[Bow78] BOWLES, Kenneth L.: A (nearly) machine independent software system for micro and mini computers. In: *SIGMINI Newsl.* 4 (1978), Nr. 1, S. 3–7. – ISSN 0163–576X

[Bro06] BROY, Manfred: Challenges in automotive software engineering. In: *ICSE '06: Proceeding of the 28th international conference on Software engineering*. New York, NY, USA : ACM Press, 2006. – ISBN 1–59593–375–1, S. 33–42

[But97] BUTTAZZO, Giorgio C.: *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Norwell, MA, USA : Kluwer Academic Publishers, 1997. – ISBN 0792399943

[But05] BUTTAZZO, Giorgio C.: Rate monotonic vs. EDF: judgment day. In: *Real-Time Syst.* 29 (2005), Nr. 1, S. 5–26. – ISSN 0922–6443

[BZ96] BENNETT, Jon C. R. ; ZHANG, Hui: WF^2Q: Worst-Case Fair Weighted Fair Queueing. In: *Proceedings of INFOCOM'96, San Francisco, CA*, 120-128

[CFH+03] CARPENTER, John ; FUNK, Shelby ; HOLMAN, Philip ; ANDERSON, James ; BARUAH, Sanjoy: *A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms*. 2003

[CH05] CHAPMAN, Matthew ; HEISER, Gernot: *Implementing Transparent Shared Memory on Clusters – Using Virtual Machines*. Proceedings of USENIX 05: General Track Anaheim, CA, USA, April 2005

[Chi07] CHISNALL, David ; L.TAUB, Mark (Hrsg.): *The Definitive Guide to the Xen Hypervisor*. Prentice Hall, 2007

[CJP07] CHAPMAN, Barbara ; JOST, Gabriele ; PAS, Ruud van d.: *Using OpenMP*. MIT Press, 2007

[Cre81] CREASY, Robert J.: The Origin of the VM/370 Time-Sharing System. In: *IBM Journal of Research and Development* 25 (1981), Nr. 5, S. 483–490

[CSG98] CULLER, David ; SINGH, J. P. ; GUPTA, Anoop: *Parallel Computer Architecture: A Hardware/Software Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, 1998. – ISBN 1558603433

[DC99] DUDA, Kenneth J. ; CHERITON, David R.: Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In: *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*. New York, NY, USA : ACM Press, 1999. – ISBN 1–58113–140–2, S. 261–276

[Ess90] ESSICK, Raymond B.: An Event-Based Fair Share Scheduler, 1990, S. 147–162

[FBYR88] FORIN, Ro ; BARRERA, Joseph ; YOUNG, Michael ; RASHID, Richard: *Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach.* 1988

[Fis07] FISHER, Nathan W.: *The Multiprocessor Real-Time Scheduling of General Task Systems*, University of North Carolina at Chapel Hill, Diss., 2007

[Fly72] FLYNN, M.: Some Computer Organizations and Their Effectiveness. In: *IEEE Transactions in Computers* C-21 (1972), S. 948–960

[Foh99] FOHLER, Gerhard: Flexible Reliable Timing - Real-Time vs. Reliability. In: *Keynote Address, 10th European Workshop on Dependable Computing*

[FR86] FITZGERALD, Robert ; RASHID, Richard F.: The integration of virtual memory management and interprocess communication in Accent. In: *ACM Trans. Comput. Syst.* 4 (1986), Nr. 2, S. 147–177. – ISSN 0734–2071

[Fra08] FRASER, Keir: *XEN Hypervisor Lines Of Code*. Xen developers Mailing list, Sept 2008. – http://lists.xensource.com/archives/html/xen-devel/2008-09/msg00355.html

[Fri08] FRIEBEL, Thomas: *Preventing Guests from spinning Around*. Xen Summit 2008, Boston MA, Juni 2008

[GGV96] GOYAL, Pawan ; GUO, Xingang ; VIN, Harrick M.: A hierarchical CPU scheduler for multimedia operating systems. In: *OSDI*. San Francisco, CA, USA, 1996, S. 107–121

[GL91] GALLMEISTER, Bill O. ; LANIER, Chris: Early experience with POSIX 1003.4 and POSIX 1003.4 A. In: *Proc Real Time Syst Symp* (1991), S. 190–198. – IEEE catalog number 91CH3090-8. ISBN 0–8186–2450–7

[Gum83] GUM, Peter H.: System/370 Extended Architecture: Facilities for Virtual Machines. In: *IBM Journal of Research and Development* 27 (1983), Nr. 6, S. 530–544

[Hei08] HEISER, Gernot: The Role of Virtualization in Embedded Systems. In: *IIES '08: Proceedings of the 1st workshop on Isolation and integration in embedded systems*. New York, NY, USA : ACM, April 2008. – ISBN 978–1–60558–126–2

[Hei09] HEISER, Gernot: *Embedded systems virtualization: Consider a Hypervisor*. Embedded.com. `http://www.embedded.com/design/212902574`. Version: January 2009

[Hen84] HENRY, G. J.: The Fair Share Scheduler. In: *AT&T Bell Laboratories Technical Journal* 63 (1984), October, Nr. 8, S. 1845–1857

[Hoa78] HOARE, C. A. R.: Communicating Sequential Processes. In: *Communications of the ACM* 21 (1978), aug, Nr. 8, S. 666–677

[HPOS05] HUBER, Bernhard ; PETI, Philipp ; OBERMAISSER, Roman ; SALLOUM, Christian E.: Using RTAI/LXRT for Partitioning in a Prototype Implementation of the DECOS Architecture. In: *3rd Workshop on Intelligent Solutions in Embedded Systems (WISES'05), Hamburg, Germany* (2005), May

[IEE02] IEEE: Standard for A Precision Clock Synchronization Protocol for Networked Measurement and Control Systems / National Institute of Standards and Technology. 2002. – IEEE

[JB06] JOHN, Tobias ; BAUMGARTL, Robert: *Worst Case Behavior of CPU Caches*. 6th Real-time Linux Workshop, Lanzhou, China, October 2006

[Kah05] KAHLE, Jim: The Cell Processor Architecture. In: *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA : IEEE Computer Society, 2005. – ISBN 0–7695–2440–0, S. 3

[Kai06] KAISER, Robert: Koexistenz unterschiedlicher Zeitanforderungen in einem gemeinsamen Rechensystem. In: HOLLECZEK, Peter (Hrsg.) ; VOGEL-HEUSER, Birgit (Hrsg.): *Echtzeitsysteme im Alltag*, Springer, November 2006 (Informatik Aktuell). – ISBN 978–3–540–47690–0, S. 16–25

[Kai08] KAISER, Robert: Estimating Context Switch Cost: A Practitioner's Approach. In: *OSPERT 2008 – Proceedings of the Fourth International Workshop on Operating Systems Platforms for Embedded Real-*

Time Applications. Praque, Czech Republic, July 2008 (UNC Technical Report TR08-010), S. 73–82

[KL88] KAY, J. ; LAUDER, P.: A fair share scheduler. In: *Commun. ACM* 31 (1988), Nr. 1, S. 44–55. – ISSN 0001–0782

[Kop91] KOPETZ, Hermann: Event-Triggered Versus Time-Triggered Real-Time Systems. In: *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond*. London, UK : Springer-Verlag, 1991. – ISBN 3–540–54987–0, S. 87–101

[Law96] LAWTON, Kevin P.: Bochs: A Portable PC Emulator for Unix/X. In: *LINUX J.* 1996 (1996), sep, Nr. 29es, S. 7. – ISSN 1075–3583

[LCC+75] LEVIN, R. ; COHEN, E. ; CORWIN, W. ; POLLACK, F. ; WULF, W.: Policy/mechanism separation in Hydra. In: *SIGOPS Oper. Syst. Rev.* 9 (1975), Nr. 5, S. 132–140. – ISSN 0163–5980

[Lie95] LIEDTKE, Jochen: On μ-Kernel Construction. In: *SOSP*, 1995, S. 237–250

[LL73] LIU, C. L. ; LAYLAND, James W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. In: *Journal of the ACM* 20 (1973), Nr. 1, S. 46–61

[LRD95] LINDSTROM, A. ; ROSENBERG, J. ; DEARLE, A.: *The grand unified theory of address spaces*. In Proceedings of the 5th Workshop on Hot Topic in Operating Systems (HotOS), pages 66–71, Orcas Island, WA, USA., May 1995

[LUC+05] LEVASSEUR, Joshua ; UHLIG, Volkmar ; CHAPMAN, Matthew ; CHUBB, Peter ; LESLIE, Ben ; HEISER, Gernot: *Pre-Virtualization: Slashing the Cost of Virtualization*. Technical Report PA005520, NICTA, October 2005

[LY99] LINDHOLM, Tim ; YELLIN, Frank: *Java Virtual Machine Specification*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1999. – ISBN 0201432943

[MDP00] MANTEGAZZA, P. ; DOZIO, E. L. ; PAPACHARALAMBOUS, S.: RTAI: Real Time Application Interface. In: *Linux J.* 2000 (2000), Nr. 72es, S. 10. – ISSN 1075–3583

[MT84] MAY, David ; TAYLOR, Richard: *Occam: An Overview*. Microprocessors and Microsystems, Vol8, No. 2, pp.73-79, March 1984

[MUKX06] MERGEN, Mark F. ; UHLIG, Volkmar ; KRIEGER, Orran ; XENIDIS, Jimi: Virtualization for high-performance computing. In: *SIGOPS Oper. Syst. Rev.* 40 (2006), Nr. 2, S. 8–11. – ISSN 0163–5980

[NVZ01] NIEH, Jason ; VAILL, Christopher ; ZHONG, Hua: Virtual-Time Round-Robin: An O(1) Proportional Share Scheduler. In: *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, 2001. – ISBN 1–880446–09–X, S. 245–259

[OB98] OH, Dong ik ; BAKER, Theodore P.: Utilization Bounds for N Processor Rate Monotone Scheduling with Static Processor Assignment. In: *Real-Time Systems* 15 (1998), S. 183–192

[Ous80] OUSTERHOUT, J. K.: *Partitioning and Cooperation in a Distributed Multiprocessor Operating System: Medusa*, Computer Science Department, Carnegie-Mellon University, Diss., apr 1980

[PG74] POPEK, Gerald J. ; GOLDBERG, Robert P.: Formal requirements for virtualizable third generation architectures. In: *Commun. ACM* 7 (1974), Nr. 7, S. 412–421

[PG93] PAREKH, Abhay K. ; GALLAGER, Robert G.: A generalized processor sharing approach to flow control in integrated services networks: the single-node case. In: *IEEE/ACM Trans. Netw.* 1 (1993), Nr. 3, S. 344–357. – ISSN 1063–6692

[PH05] PATTERSON, David A. ; HENNESSY, John L. ; BODE, Arndt (Hrsg.) ; KARL, Wolfgang (Hrsg.) ; UNGERER, Theo (Hrsg.): *Rechnerorganisation und -entwurf – Die Hardware/Software-Schnittstelle*. Elsevier, 2005

[RI00] ROBIN, J. ; IRVINE, C.: *Analysis of the Intel Pentium's ability to support a secure virtual machine monitor*. `http://citeseer.ist.psu.edu/robin00analysis.html`. Version: 2000

[Ros04] ROSE, Robert: *A Survey of Virtualization Techniques*. Mar 2004

[SAWJ96] STOICA, I. ; ABDEL-WAHAB, H. ; JEFFAY, K.: On the Duality between Resource Reservation and Proportional Share Resource Allocation / Department of Computer Science, Old Dominion University, Norfolk, VA (TR_96_19). – Forschungsbericht. `citeseer.ist.psu.edu/stoica97duality.html`

[SL03] SHA, Liu ; LEE, Chang-Gun: *Real Time Virtual Machines for Legacy Avionics Software Migration*. Proc. of the 9th International Conference on Real Time Embedded Computing Systems and Applications, 2003

[SPC03] SÁNCHEZ-PUEBLA, Miguel A. ; CARRETERO, Jesús: A new approach for distributed computing in avionics systems. In: *ISICT '03: Proceedings of the 1st international symposium on Information and communication technologies*, Trinity College Dublin, 2003, S. 579–584

[Sta05] STALLINGS, William: *Operating Systems: internals and design principles. – 5th edition*. Prentice Hall, 2005. – ISBN 0–13–147954–1

[SV95] SHREEDHAR, M. ; VARGHESE, George: Efficient Fair Queueing Using Deficit Round Robin. In: *SIGCOMM*, 231-242

[SVL01] SUGERMAN, Jeremy ; VENKITACHALAM, Ganesh ; LIM, Beng hong: *Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor*. 2001

[ULSD04] UHLIG, Volkmar ; LEVASSEUR, Joshua ; SKOGLUND, Espen ; DANNOWSKI, Uwe: Towards Scalable Multiprocessor Virtual Machines. In: *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, 2004

[VMw05a] VMWARE, INC: *Best Practices using VMware Virtual SMP*. VMware White Paper, 2005

[VMw05b] VMWARE, INC.: *VMware ESX Server Online Documentation*. `http://www.vmware.com`. Version: 2005

[VMw07] VMWARE, INC.: *A Performance Comparison of Hypervisors*. March 2007

[Wal95] WALDSPURGER, C. A.: Lottery and Stride Scheduling: Flexible Proportional-share Resource Management / MIT Laboratory for Computer Science Cambridge, MA 02139 USA (MIT/LCS/TR-667). – Forschungsbericht. – 151 S. `citeseer.ist.psu.edu/waldspurger95lottery.html`

[Wei84] WEICKER, Reinhold P.: Dhrystone: a synthetic systems programming benchmark. In: *Commun. ACM* 27 (1984), Nr. 10, S. 1013–1030. – ISSN 0001–0782

[WSG02] WHITAKER, A. ; SHAW, M. ; GRIBBLE, S.: *Denali: Lightweight virtual machines for distributed and networked applications*. 2002

[Xen05] XENSOURCE INC.: *Xen User' Manual v3.0*. 2005

[YATR+87] YOUNG, Michael ; AVADIS TEVANIAN, Jr. ; RASHID, Richard ; GOLUB, David ; EPPINGER, Jeffrey ; CHEW, Jonathan ; BOLOSKY, William ; BLACK, David ; BARON, Robert: *The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System*. Proceedings of the 11th Symposium on Operating Systems Principles, November 1987

[Zö8] ZÖBEL, Dieter: *Echtzeitsysteme - Grundlagen der Planung*. exam.press. Springer-Verlag, Berlin, 2008

[ZA95] ZÖBEL, Dieter ; ALBRECHT, Wolfgang: *Echtzeitsysteme-Grundlagen und Techniken*. 1. Attenkirchen : International Thomson Publishing, 1995

[Zha91] ZHANG, L.: Virtual clock: A new traffic control algorithm for packet-switched networks. In: *ACM Trans. Computer Systems, 9: 101-124, 1991.* 9 (1991), May., S. 101–124