Autonomic- und Organic-Computing-Techniken für eingebettete Echtzeitsysteme

Dissertation

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

der Fakultät für Angewandte Informatik

der Universität Augsburg

eingereicht von

Dipl.-Inf. Florian Kluge

Erstgutachter:	Prof. Dr. rer. nat. Theo Ungerer
Zweitgutachter:	Prof. Dr.-Ing. Rudi Knorr

Tag der mündlichen Prüfung: 26. Juli 2010

Zusammenfassung

Eingebettete Systeme werden heute in einer Vielzahl technischer Geräte zu deren Steuerung eingesetzt, wobei sie häufig unter Echtzeitanforderungen arbeiten. Durch ihren Einsatz können solche Geräte den steigenden Anforderungen etwa hinsichtlich der Sicherheit oder Energieeffizienz gerecht werden, was aber dazu führt, dass eingebettete Systeme immer komplexer werden. Eine wesentliche Rolle für diese Komplexitätssteigerung spielt die zunehmende Vernetzung eingebetteter Steuereinheiten, so dass diese Systeme immer schwieriger zu beherrschen sind. Dem wirkt man entgegen, indem man die einzelnen Komponenten eines solchen eingebetteten Netzes möglichst statisch konfiguriert und ihnen nur wenige Freiheitsgrade gibt. Der Ansatz, eine Steuereinheit für nur eine Aufgabe zu verwenden, erleichtert zwar deren Entwicklung, wird aber für die Zukunft nicht mehr zielführend sein. Eine Dynamisierung von eingebetteten Systemen ist also unausweichlich, wird aber durch die dort oft herrschenden Echtzeitanforderungen stark erschwert.

Einen Ausweg aus dieser Lage bieten der Einsatz von modernen mehrfädigen Prozessorarchitekturen sowie die Paradigmen des *Autonomic Computing* und *Organic Computing*. Autonomic und Organic Computing zielen darauf ab, den Administrationsaufwand von komplexen Computersystemen zu reduzieren. Stattdessen sollen Computersysteme sich möglichst selbst organisieren und auf Problemsituationen möglichst sinnvoll reagieren. Zukünftige Computersysteme sollen dazu die Selbst-X-Fähigkeiten Selbstkonfiguration, Selbstheilung, Selbstoptimierung und Selbstschutz implementieren. Mehrfädige Prozessoren stellen hier eine technische Möglichkeit für den Einsatz der Selbst-X-Fähigkeiten in eingebetteten Systemen dar. Sie verfügen über genug Rechenkapazität, um über die eigentliche Anwendung hinaus parallel noch weitere Programme ausführen zu können, wobei sie durch die mehrfädige Programmausführung trotzdem den Echtzeitanforderungen eingebetteter Systeme gerecht werden können. All das setzt aber auch einen durchgehenden Software-Entwurf voraus, der die Anforderungen aus den Bereichen Echtzeit sowie Autonomic und Organic Computing beachtet.

Diese Arbeit stellt einen solchen Software-Entwurf auf Ebene einer eingebetteten Steuereinheit vor und berücksichtigt dabei bereits eine mögliche Vernetzung von Steuereinheiten. Darüber hinaus wird ein Echtzeitbetriebssystem vorgestellt, das auch den Anforderungen des Autonomic und Organic Computing gerecht wird. Darauf aufbauend werden die Architektur und Implementierung eines *Autonomic Management* präsentiert, das die besonderen Anforderungen von Echtzeitsystemen berücksichtigt. Dazu werden nicht echtzeitfähiger Reaktionen von der Echtzeitanwendungen zeitlich getrennt ausgeführt. Evaluierungen des Gesamtsystems

auf einem simultan mehrfädigen Prozessor zeigen die Praktikabilität des Ansatzes und die damit erzielten Verbesserungen.

Vorwort

Diese Arbeit entstand in den Jahren 2005 bis 2010 während meiner Tätigkeit als wissenschaftlicher Angestellter am Lehrstuhl für Systemnahe Informatik und Kommunikationssysteme der Universität Augsburg. Im Rahmen des DFG-Projekts *CAR-SoC* entwarf ich eine Systemsoftware für einen mehrfädigen eingebetteten Prozessor, die verschiedene Aspekte des Autonomic Computing integriert. Dabei entstand das Ziel, die Implementierung dieser Aspekte zu vereinheitlichen, was diese Arbeit wesentlich beeinflusst hat.

An dieser Stelle möchte ich mich bei Herrn Prof. Theo Ungerer für die erstklassige Betreuung und die vielen Diskussionen bedanken, die diese Arbeit erst ermöglicht haben. Bei Herrn Prof. Rudi Knorr bedanke ich mich für die Übernahme des Zweitgutachtens. Herrn Prof. Uwe Brinkschulte danke ich für die fruchtbaren Diskussionen während unserer Projekttreffen. Weiterhin danke ich meinen Kollegen am Lehrstuhl für die gemeinsamen Diskussionen, die zum Gelingen dieser Arbeit beigetragen haben. Sascha Uhrig und Jörg Mische möchte ich auch für die gemeinsame Arbeit am CAR-SoC-Projekt danken. Mein Dank geht auch an Stefan Schenk sowie meine Eltern für das Korrekturlesen dieser Arbeit. Meiner Familie und meinen Freunden möchte ich für ihre Unterstützung und die Zeit miteinander danken, die mir den nötigen Ausgleich und Entspannung brachte.

Augsburg, 29. Juli 2010 *Florian Kluge*

Inhaltsverzeichnis

1 Einleitung

Die meisten technischen Geräte verfügen heutzutage über in sie eingebettete Computer, sogenannte *eingebettete Systeme*. Solche eingebetteten Systeme erfüllen verschiedenste Aufgaben. Sie können der Steuerung oder Überwachung des sie umgebenden Geräts dienen, oder auch Aufgaben aus dem Bereich der Daten- oder Signalverarbeitung erfüllen. Die Einsatzfelder von eingebetteten Systemen sind sehr vielfältig. So finden sich eingebettete Systeme heute etwa in Haushaltsgeräten wie Waschmaschinen, Kommunikations- und Unterhaltungsgeräten wie Mobiltelefonen, aber auch in Kraftfahrzeugen und Flugzeugen. Dem Nutzer solcher Geräte ist das Vorhandensein eingebetteter Systeme oftmals verborgen. Durch die Einbettung in den Gerätekontext ergeben sich für eingebettete Systeme gegenüber solchen als Computer erkennbaren PC- oder Server-Systemen üblicherweise einige Einschränkungen, etwa hinsichtlich ihrer Kosten oder ihres Platzverbrauchs. Auch müssen sie mit wenig Speicher auskommen sowie, gerade bei mobilen Geräten, einen geringen Energieverbrauch aufweisen. Diesen Einschränkungen begegnet man dadurch, dass eingebettete Systeme bei ihrem Entwurf auf die jeweilige Aufgabe abgestimmt werden. Oft kommen hier auch spezielle Prozessoren, sogenannte Einchipsysteme (*System-On-a-Chip, SoC*), zum Einsatz, die neben dem eigentlichen Prozessor auch Bus, Speicher und Peripherie auf einem Chip integrieren. Eingebettete Systeme arbeiten häufig unter Echtzeitanforderungen. Hier ist nicht eine durchschnittliche hohe Verarbeitungsgeschwindigkeit relevant, sondern eine zeitlich vorhersagbare Ausführung der Anwendungen. Damit stellt man sicher, dass das eingebettete System innerhalb vorgegebener Zeitschranken auf Ereignisse reagieren kann.

Eingebettete Systeme werden zunehmend nicht mehr alleinstehend betrieben, sondern die einzelnen Einheiten sind oftmals miteinander vernetzt. Heutige Fahrzeuge etwa können bis zu 80 eingebettete Steuereinheiten (*Embedded Control Units, ECUs*) enthalten. Diese erfüllen verschiedenste Aufgaben von der Motorsteuerung bis hin zur Medienwiedergabe. Verbunden sind all diese Einheiten über mehrere Kommunikationsnetze, typischerweise CAN (*Controller Area Network*) und FlexRay. Ähnliche Netze finden sich in anderen Transportmitteln wie etwa Flugzeugen, aber auch in großen Industrieanlagen. Diese Vernetzung der Steuereinheiten kann aber zu Problemen führen. Durch die Interaktion der Steuereinheiten untereinander können neue Fehler entstehen, die bei deren Einzelentwurf noch nicht absehbar waren. Aufgrund dieser höheren Komplexität wird

die Fehlersuche in solche einem System erheblich erschwert. Ein weiteres Problem stellt die starke Spezialisierung der Steuereinheiten dar. Einzelne ECUs sind für eine bestimmte Funktionalität entwickelt. Fällt eine solche ECU innerhalb des Netzes aus, so geht auch deren Funktionalität verloren. Gleichzeitig aber verfügen die ECUs aufgrund ihrer Auslegung für den längstmöglichen Ausführungspfad oft über freie Rechenkapazitäten. Falls dieser aber selten genommen wird, liegen die überschüssigen Rechenkapazitäten die meiste Zeit brach.

Einen möglichen Ausweg aus dieser Problematik bieten das Paradigma des *Organic Computing* und moderne mehrfädige Prozessorarchitekturen. *Organic Computing* fasst Techniken zusammen, durch die insbesondere eingebettete Systeme selbstkonfigurierend, selbstheilend, selbstoptimierend und selbstschützend werden sollen. Durch diese sogenannten Selbst-X-Fähigkeiten sollen sich informationstechnische Systeme wie lebende Organismen selbstorganisierend verhalten. Die wachsende Komplexität dieser Systeme soll damit beherrscht und deren Verhalten sogar verbessert werden. Moderne mehrfädige Prozessoren eröffnen durch ihre gesteigerte Rechenleistung die Möglichkeit, mehr und komplexere Anwendungen auszuführen, und parallel dazu das Gesamtsystem mithilfe von Techniken des Organic Computing zu kontrollieren.

1.1 Ziele dieser Arbeit

Die vorliegende Arbeit greift das Problem der wachsenden Komplexität eingebetteter Systeme auf. Sie soll zeigen, wie Konzepte des *Autonomic Computing* und *Organic Computing* in solche Systeme integriert werden können, so dass diese beherrschbar bleiben. Ebenso soll sie mit Hilfe dieser Konzepte eine effiziente Ausnutzung einzelner Steuereinheiten erreichen und die Ausfallsicherheit von Diensten erhöhen. Eine besondere Herausforderung sind die Echtzeitanforderungen, unter denen eingebettete Systeme häufig arbeiten. Hier gilt es sicherzustellen, dass die verwendeten Selbstorganisationstechniken gegebenenfalls laufende Echtzeitanwendungen in keinem Fall negativ beeinflussen können. Die Nutzung geeigneter Prozessorarchitekturen ist dafür eine wichtige Voraussetzung.

Diese Arbeit soll zeigen, welche konkreten Anforderungen durch den Einsatz von Autonomic- und Organic-Computing-Techniken in dem Umfeld von eingebetteten Echtzeitsystemen entstehen. Auf Basis dieser Anforderungen sollen ein Echtzeitbetriebssystem sowie ein *Autonomic Management* für eingebettete Echtzeitsysteme entworfen werden. Das Echtzeitbetriebssystem garantiert dabei nicht nur die korrekte Ausführung von Echtzeitanwendungen, sondern bringt auch die nötigen Fähigkeiten mit, um erfolgreich Techniken des Autonomic oder Organic Computing in einer Steuereinheit zu integrieren. Die Steuereinheit soll so mit Selbst-X-Fähigkeiten ausgestattet werden. Als zugrundeliegende Hardware

soll ein simultan mehrfädiger Prozessor dienen, der an heute in eingebetteten Echtzeitsystemen genutzte Prozessoren angelehnt ist. Dieser verfügt über Leistungsreserven, die es erlauben, Selbst-X-Techniken auszuführen ohne laufende Echtzeitanwendungen zu beeinträchtigen.

1.2 Aufbau dieser Arbeit

Das nächste Kapitel befasst sich mit den Grundlagen für diese Arbeit. Es führt in die Problematik von Echtzeitsystemen und mehrfädigen Mikroprozessoren ein und stellt Architekturen des Autonomic und Organic Computing vor. Zu all diesen Punkten definiert es Anforderungen an die Software. Abschließend wird der Stand der Technik betrachtet, der auf die vorher definierten Anforderungen untersucht wird. Kapitel 3 stellt die Architektur und Implementierung eines Echtzeitbetriebssystems für eingebettete Systeme vor. Diese Architektur berücksichtigt insbesondere die im vorgehenden Kapitel definierten Anforderungen hinsichtlich des Organic Computing. Kapitel 4 stellt eine Management-Architektur zur Implementierung von Organic-Computing-Techniken in eingebetteten Echtzeitsystemen vor. Für diese wird in Kapitel 5 ein generischer Algorithmus zur Entscheidungsfindung entworfen und implementiert. Kapitel 6 vervollständigt die in den beiden vorhergehenden Kapiteln präsentierte Architektur mit Beispielen für anwendungsspezifische Managementkomponenten. Das auf diese Weise entstandene System wird in Kapitel 7 evaluiert. Dabei werden sowohl einzelne Managementkomponenten als auch deren Zusammenarbeit mit dem generischen Management aus Kapitel 5 untersucht. Kapitel 8 fasst die Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick auf weitere mögliche Forschungsarbeiten.

2 Grundlagen

Dieses Kapitel betrachtet die technischen Grundlagen für diese Arbeit. Es führt zunächst das Feld eingebetteter Echtzeitsysteme ein. Daran schließt sich ein Überblick über mehrfädige Prozessoren an, der auch den für diese Arbeit verwendeten simultan mehrfädigen Prozessor vorstellt. Als dritte Grundlage werden die Konzepte des *Autonomic* und *Organic Computing* erläutert. Abschließend untersucht es heutige Echtzeitbetriebssysteme auf ihre Fähigkeiten hinsichtlich des Organic Computing.

2.1 Eingebettete Echtzeitsysteme

Als *Eingebettetes System* bezeichnet man einen Computer, der in ein technisches Gerät eingebunden ist. Der Computer nimmt dabei Aufgaben der Steuerung, Regelung oder Überwachung wahr. Falls diese Aufgaben unter Echtzeitbedingungen erfüllt werden, spricht man von einem eingebetteten *Echtzeitsystem*. Dabei wird zwischen zwei grundlegenden Arten von Echtzeitanforderungen unterschieden [33]:

- **Harte** Echtzeitanforderungen: das Ergebnis einer Berechnung *muss spätestens* bis zu einem bestimmten Zeitpunkt vorliegen. Die Folgen beim Verpassen dieser Zeitschranke sind im Allgemeinen schwerwiegend. Zum Beispiel unterliegt die Auslösung des Airbags im Auto bei einem Aufprall harten Echtzeitanforderungen.
- **Weiche** Echtzeitanforderungen: das Ergebnis einer Berechnung *soll* bis zu einem bestimmten Zeitpunkt vorliegen. Falls diese Zeitschranke überschritten wird, so wird das Ergebnis zwar verwendet, allerdings verringert sich sein Nutzen. Die Dekodierung eines Videostroms etwa unterliegt weichen Echtzeitbedingungen. Falls ein Einzelbild zu spät bereitsteht, leidet zwar die Filmqualität darunter, aber der Film selbst kann trotzdem weiter dargestellt werden.

Um die Einhaltung der Zeitschranken (*Deadlines*) garantieren zu können, muss es möglich sein, für die zeitkritischen Programmabschnitte eine maximale Laufzeit anzugeben (*Worst Case Execution Time, WCET*). Diese muss geringer sein als

die Zeitschranke, die durch den Einsatzzweck vorgegeben ist. Ein Echtzeitsystem zeichnet sich also insbesondere durch die klare Analysierbarkeit seines Zeitverhaltens aus. Die tatsächliche Verarbeitungsgeschwindigkeit ist zweitrangig. Die Ausführungsplattform muss aber in Abhängigkeit von der WCET so dimensioniert werden, dass die Einhaltung der Zeitschranken garantiert ist. Ein weiteres Ziel ist es, dass die tatsächliche Ausführungszeit möglichst wenig von der berechneten WCET abweicht. Damit lässt sich eine übermäßige Dimensionierung der Ausführungsplattform vermeiden, was Produktionskosten einspart. Zur Bestimmung der WCET gibt es zwei Herangehensweisen:

- Bei der *statischen WCET-Analyse* wird der längste Ausführungspfad eines Programms bestimmt. Dessen Laufzeit wird mittels Programmfluss- und Maschinencodeanalyse bestimmt. In diese Berechnungen gehen auch die Besonderheiten der zugrunde liegenden Prozessorarchitektur ein.
- Bei der *messungsbasierten WCET-Analyse* hingegen wird das Programm instrumentiert und auf der Zielhardware ausgeführt. Die Instrumentierung erzeugt bei der Ausführung Zeitstempel, mit deren Hilfe die Ausführungszeit bestimmt werden kann. Die Ausführungszeit hängt dabei generell von den Eingabedaten ab. Durch eine Analyse des Programmcodes lassen sich Eingabedatensätze erzeugen, mit denen alle Programmpfade abgedeckt werden.

Aus der WCET-Analyse ergeben sich dabei die Mindestanforderungen für die Prozessorauswahl, aber auch neue Möglichkeiten für Optimierungen in der Software. Harte Echtzeitsysteme werden dabei für den schlechtesten Lastfall ausgelegt, so dass auch bei einem Programmpfad mit maximaler Länge alle Zeitschranken eingehalten werden. Im Normalfall sind hier die Prozessoren nicht voll ausgelastet. Auch die Hardware muss dabei so ausgelegt sein, dass Unvorhersagbarkeiten bei der Laufzeitanalyse ausgeschlossen werden. Dies führt zu Einschränkungen bei der Nutzung von leistungssteigernden Techniken, wie sie bei *General-Purpose-Prozessoren* verbreitet sind. So können etwa Caches nur dann verwendet werden, wenn einzelne Bereiche ständig exklusiv für den kritischen Prozess reserviert sind. Andernfalls müssten bei einer WCET-Analyse für alle Speicherzugriffe Cache-Misses angenommen werden, so dass sich für die WCET keinerlei Straffung ergäbe. Aus ähnlichen Gründen kann auf die Verwendung spekulativer Ausführungstechniken in der Prozessorpipeline verzichtet werden.

Die Programmierung von Echtzeitsystemen erfordert besondere Umsicht. Hier sind drei Ziele zu erreichen:

1. die Ausführungszeit von Programmen muss vorhersehbar und begrenzt sein,
2. das Programm muss statisch WCET-analysierbar sein, und

3. die berechnete WCET soll möglichst straff, also keine zu pessimistische Abschätzung sein.

Dies bedeutet vor allem, dass man auf viele Techniken und Konzepte verzichten muss, die in General-Purpose-Systemen selbstverständlich eingesetzt werden. Das Hauptziel ist es, die Dynamik der Ausführungspfade zu minimieren, also den Programmfluss möglichst von Verzweigungen (`if/else`) freizuhalten. An Stellen, an denen dies nicht möglich ist, sollen die einer Verzweigung folgenden Programmblöcke möglichst ähnliche Komplexität besitzen. So wird eine Überschätzung der WCET vermieden, die sich immer an dem längstmöglichen Pfad orientiert. Ebenso schädlich ist der Einsatz von Rekursionen oder Schleifen, deren Grenzen erst zur Laufzeit bestimmt werden. Auf Algorithmen, deren Zeitverhalten von vorherigen Aufrufen abhängt oder die ein völlig nichtdeterministisches Zeitverhalten haben, muss weitgehend verzichtet werden. Damit ist unter anderem zumeist die Nutzung einer dynamischen Speicherverwaltung ausgeschlossen, wenn sie nicht auf speziellen, echtzeitfähigen Algorithmen basiert. Falls mehrere Echtzeitprozesse untereinander Informationen austauschen müssen, so muss auch diese Kommunikation vollständig deterministisch sein.

All diese Einschränkungen schlagen sich auch darin nieder, dass für Echtzeitsysteme spezielle Betriebssysteme existieren. Die Schedulingverfahren moderner General-Purpose-Betriebssysteme erlauben typischerweise keine Vorhersage über den zeitlichen Ablauf der Taskausführungen. Für Echtzeitsysteme werden daher spezielle Echtzeitbetriebssysteme (*Real-Time Operating System, RTOS*) eingesetzt, die geeignete Schedulingverfahren implementieren. Weiterhin gewährleisten diese Echtzeitbetriebssysteme eine weitgehende Isolation gleichzeitig laufender Anwendungen voneinander und bieten deterministische Kommunikationsmechanismen für die Interprozesskommunikation. Damit wird eine gegenseitige Beeinflussung minimiert, was wiederum die WCET-Analyse erleichtert und außerdem die Sicherheit des Gesamtsystems erhöht.

Bei weichen Echtzeitsystemen sind die Anforderungen prinzipiell ähnlich. Da man aber in Kauf nimmt, dass gelegentlich Deadlines verpasst werden, bestehen hier mehr Optimierungsmöglichkeiten. Dadurch lassen sich bei der Auswahl und Ausnutzung der Hardware unter Umständen Kosten einsparen.

2.1.1 Anforderungen an ein Echtzeitbetriebssystem

Die Aufgabe eines Betriebssystems ist es, die Arbeit an einem Computersystem zu ermöglichen. Es steuert die Programmausführung und verwaltet Betriebsmittel wie Speicher sowie Ein- und Ausgabegeräte. Grundlegende Sicherheitsfunktionalitäten stellt ein Betriebssystem in Form verschieden privilegierter Ausführungs-

modi bereit. Im *Kernel Mode* ist ein freier Zugriff auf alle Hardware-Ressourcen möglich, während im *User Mode* gewisse Einschränkungen gelten.

Bei einem *Echtzeitbetriebssystem* handelt es sich um ein Betriebssystem, das für die Ausführung von Anwendungen mit Zeitschranken konzipiert ist. Es stellt ein vorhersagbares Zeitverhalten zur Verfügung und garantiert das Einhalten der Zeitschranken.

Hierzu lassen sich folgende Mindestanforderungen an ein Echtzeitbetriebssystem aufstellen:

RTOS-1 (Mehrfädigkeit) Das Betriebssystem ist **mehrfädig** und **unterbrechbar** (präemptiv). Die Ausführung eines Tasks oder einer Betriebssystemfunktion kann jederzeit zu Gunsten eines Tasks mit höherer Priorität unterbrochen werden. Dadurch stellt das Betriebssystem sicher, dass die Anwendungsplattform auf Ereignisse innerhalb fester Zeitschranken reagieren kann.

RTOS-2 (Echtzeitscheduling) Das Betriebssystem verfügt über einen **echtzeitfähigen Scheduler**. Dieser stellt sicher, dass die Applikationen vorgegebene Zeitschranken einhalten können. Im einfachsten Fall implementiert das Betriebssystem zum Beispiel ein Prioritätenscheduling. Aber auch der Einsatz von zeitbasierten Schedulingtechniken wie etwa *Earliest Deadline First* oder *Least Laxity First* (EDF, LLF) ist möglich [31].

RTOS-3 (Prozesssynchronisation) Das Betriebssystem unterstützt **vorhersagbare Synchronisationsmechanismen**, die neben einem deterministischen Verhalten eine maximale Verweildauer in kritischen Bereichen gewährleisten. Beim Zugriff mehrerer Tasks auf gemeinsam genutzte Ressourcen ermöglicht es die Angabe einer maximalen Wartezeit für jeden Task, falls die Ressource aktuell durch einen anderen Task belegt ist. Außerdem stellen die Synchronisationsmechanismen Mittel zur Vermeidung von Prioritätsinversion und/oder Verklemmungen bereit.

RTOS-4 (Echtzeitverhalten) Das **Verhalten** des Betriebssystems ist bekannt. Dies betrifft insbesondere Interrupt-Latenzen sowie die maximalen Ausführungszeiten von Systemaufrufen. Diese müssen beschränkt, vorhersagbar und unabhängig von der aktuellen Systemlast sein.

Ziel dieser Anforderungen ist es, bereits auf Betriebssystemebene ein vorhersagbares Zeitverhalten bereitzustellen und den Anwendungsentwurf in dieser Hinsicht zu unterstützen.

Ein weit verbreitetes Merkmal von Echtzeitbetriebssystemen ist eine Mikrokern-Architektur (*Micro Kernel*, μ-Kern). Ein solcher Mikrokern verfügt im Gegensatz zu dem im Desktop-Bereich verbreiteten *monolithischen Kernel* (z.B.

GNU/Linux, OS/2) nur über grundlegende Funktionalitäten. Darunter fallen etwa die Speicher- und Prozessverwaltung mit Kommunikations- und Synchronisationsmechanismen. Darüber hinausgehende Funktionen wie etwa Gerätetreiber oder der Zugriff auf Dateisysteme werden als eigene Programmbibliotheken implementiert.

Dies hat den Vorteil, dass einzelne Komponenten des Betriebssystems, aber auch der Anwendungen, separiert sind und ausgetauscht werden können, ohne andere Teile des Systems zu beeinträchtigen. Insbesondere führt der Ausfall einer Komponente nicht zwangsläufig zum Ausfall des Gesamtsystems. Gerätetreiber werden bei einer μ-Kern-Architektur im nicht-privilegierten Benutzermodus ausgeführt. Dadurch ergibt sich eine bessere Kontrolle der Zugriffsrechte und somit ein deutlich geringeres Sicherheitsrisiko. Außerdem erhöht die starke Kapselung der einzelnen Komponenten die Analysierbarkeit des Gesamtsystems.

Eine μ-Kern-Architektur bringt aber auch einige Nachteile mit sich. So kommt es durch den häufigen Wechsel zwischen privilegiertem und Benutzermodus zu ebenso häufigen Kontextwechseln. Diese sind oft mit einem erheblichen Zeitaufwand verbunden. Auch gestaltet sich die Synchronisation der einzelnen Nutzerprozesse oftmals als schwierig, da diese komplexe Kernel-Prozesse benötigt. Ähnlich verhält es sich sich auch mit Hardware-Zugriffen, da diese aus dem nichtprivilegierten Benutzermodus heraus typischerweise nicht realisierbar sind. Stattdessen müssen hier auch spezielle Betriebssystemaufrufe bereitgestellt werden.

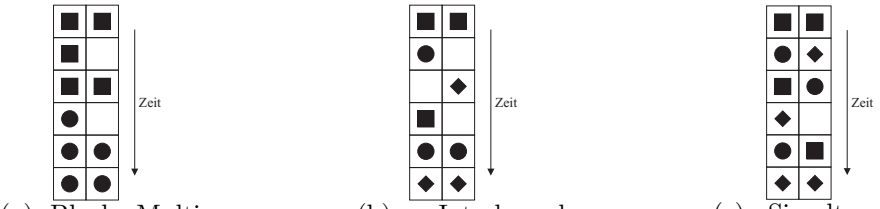
2.2 Mehrfädige Mikroprozessoren

Mehrfädige Mikroprozessoren können mit nur einem Prozessorkern mehrere Programme quasi parallel bearbeiten. Dazu besitzen sie für jeden Thread einen eigenen Registersatz und Programmzähler. Von ihrer Hardwarekomplexität sind sie zwischen herkömmlichen einfädigen Prozessoren und Mehrkernprozessoren anzusiedeln. Durch die Mehrfädigkeit ergibt sich eine bessere Ausnutzung der gesamten CPU. Zwar sind moderne Prozessoren bereits mit verschiedenen Techniken wie etwa Sprungvorhersage ausgestattet, um die Pipeline-Auslastung zu erhöhen, es hat sich aber gezeigt, dass durch mehrfädige Programmausführung weitere Steigerungen möglich sind.

- Beim **Block Multithreading** [1] weist der Prozessor Befehle eines Threads über mehrere Takte der Pipeline zu (Abbildung 2.1(a)). Am Ende eines solchen Blocks bestimmt der Scheduler, welcher Thread als nächstes ausgeführt wird. Üblicherweise wird ein Block dann beendet, wenn sich für diesen Thread aufgrund eines Ereignisses eine hohe Wartezeit ergibt, etwa bei der synchronen Kommunikation in einem Netz. Die dadurch entste-

hende Latenz wird nach einem Wechsel des Prozessorkontexts durch die Ausführung eines anderen Threads ausgenutzt (*switch-by-event*). Manche Architekturen erlauben es auch, einen Block nach einer festen Anzahl Takten zu beenden (*time-slice*). In Abbildung 2.1(a) ist ein Wechsel zwischen zwei Threads ohne jegliche Latenz dargestellt. Je nach Architektur kann dieser Wechsel aber auch mehrere Takte Latenz erzeugen, während derer der Thread-Kontext ausgetauscht wird.

- Beim **Interleaved Multithreading** [29] weist der Prozessor jeden Takt eine oder mehrere Instruktionen eines anderen Threads der Pipeline zu (Abbildung 2.1(b)). Der Prozessor muss hierbei einen Kontextwechsel ohne jegliche Latenz erlauben. Idealerweise wird erst dann wieder ein Befehl eines bestimmten Threads zugewiesen, wenn der vorherige die Pipeline passiert hat. Auf diese Weise lassen sich Pipeline-Konflikte vermeiden. Um beim Interleaved Multithreading eine effiziente Nutzung der Pipeline zu erzielen, müssen ausreichend Threads zur Ausführung bereitstehen.
- **Simultaneous Multithreading** [60] wurde als Erweiterung zur Leistungssteigerung in superskalaren Prozessoren entworfen. Beim Simultaneous Multithreading (SMT) kann der Prozessor in jedem Takt Befehle verschiedener Threads der Pipeline zuweisen (Abbildung 2.1(c)). Damit lässt sich die höchste Leistungssteigerung erzielen.



(a) Block Multithreading (b) Interleaved Multithreading (c) Simultaneous Multithreading

Abbildung 2.1: Konzepte zum Hardware-Multithreading am Beispiel einer zweifach-superskalaren Pipeline

2.2.1 Anforderungen für den Echtzeitbetrieb

Um ihn für Echtzeitanwendungen einsetzen zu können, muss ein mehrfädiger Prozessor eine vorhersagbare Programmausführung erlauben. Bezüglich einfädiger Programmausführung unterscheidet er sich damit nicht von einem herkömmlichen einfädigen Prozessor. Laufen allerdings mehrere Threads parallel ab und müssen wenigstens für einen Thread Echtzeitgarantien gegeben werden, so sind einige Punkte zu beachten:

- **Threadscheduling:** Bei der Zuweisung von Instruktionen dürfen Echtzeit-Threads nicht unvorhersagbar durch parallel laufende Threads beeinflusst werden. Der Scheduler muss so arbeiten, dass vorab eine WCET-Analyse der Anwendung möglich ist. Dies stellt insbesondere beim Simultaneous Multithreading ein Problem dar [37].
- **Nutzung der Ausführungseinheiten:** Bei mehrfädigen Prozessoren ist zu beachten, dass zwar der Registersatz repliziert sein kann, sich aber alle laufenden Threads die vorhandenen Ausführungseinheiten teilen müssen. Falls eine solche Ausführungseinheit durch eine Instruktion eines niederprioren Threads für mehrere Takte belegt wird, so kann dieser unter Umständen hochpriore Threads blockieren. Beim Prozessorentwurf ist daher darauf zu achten, dass solche Blockierungen in fester Zeit beendet werden, um eine vorhersagbare Programmausführung zu gewährleisten. Eine mögliche Lösung hierfür ist etwa die Einführung von unterbrechbaren Microcodes für komplexe Operationen [61].
- **Programmabhängigkeiten auf höherer Ebene:** Durch die feinkörnig parallele Ausführung von Threads können auch auf höheren Programmebenen Probleme auftreten. Genannt sei hier insbesondere die Synchronisation zwischen zwei Threads. Für einfädige Programmausführung existieren hier Techniken, die solche kritischen Bereiche WCET-analysierbar machen, etwa das *Priority Ceiling Protocol* [56]. Bei echt paralleler Ausführung von Programmen sind diese Techniken aber, wenn überhaupt, nur noch eingeschränkt anwendbar [27]. Der Entwickler muss dies beim Programmentwurf berücksichtigen.

2.2.2 Helper Threads

Ursprünglich wurde das Konzept der *Helper Threads* für zukünftige mehrfädige Hochleistungsprozessoren eingeführt. Dabei werden in schneller Folge neue Threads erzeugt, die zeitgleich mit der Hauptanwendung ausgeführt werden. Sie können helfen, die Ausführung der Hauptanwendung zu beschleunigen. Solche Helper Threads wurden bereits zur Sprungvorhersage [10] und zur Vorhersage von Speicheradressen [11, 32, 67] vorgeschlagen. Ebenso werden sie zur Ausnahmebehandlung [19, 66] und zum beschleunigten Ausführen von Schleifen [34] genutzt.

In mehrfädigen Prozessoren können Helper Threads die Pipeline-Latenzen eines Anwendungsthreads ausnutzen. Sie werden insofern zeitlich komplett isoliert ausgeführt. Dadurch eigenen sie sich gerade auch für den Einsatz in Echtzeitsystemen. Die eingebetteten Java-Mikrocontroller Komodo [45] und jamuth [62] setzen Helper Threads zur Echtzeit-Speicherbereinigung und dem dynamischen

Laden von Softwareaktualisierungen harter Echtzeit-Threads [44] ein. In Infineons mehrfädigem Mikrocontroller TriCore 2 kann der Kontextwechsel innerhalb des Betriebssystems durch einen Helper Thread beschleunigt werden [23].

Helper Threads werden auch im CAR-SoC-Projekt genutzt, in dessen Rahmen diese Arbeit entstanden ist und das im nächsten Abschnitt erläutert wird.

2.2.3 Das CAR-SoC-Projekt

Das Forschungsprojekt CAR-SoC (**C**onnective **A**utonomic **R**eal-Time **S**ystem **o**n a **C**hip) beschäftigt sich mit der Entwicklung eines neuartigen System-on-a-Chip (SoC), das die Anforderungen von harten Echtzeitanwendungen mit einer mehrfädigen Prozessorarchitektur verbindet. Die dazu entworfene Systemsoftware (Betriebssystem) versieht jedes einzelne CAR-SoC mit Fähigkeiten aus den Bereichen des *Autonomic Computing* (AC) beziehungsweise *Organic Computing* (OC), um die steigende Komplexität solcher eingebetteten Systeme zu bewältigen. Hierauf aufbauend ermöglicht eine Middleware, mehrere CAR-SoC-Knoten untereinander zu verknüpfen. Auch diese Middleware implementiert AC-/OC-Techniken, so dass letztendlich ein selbstorganisierendes Netz aus CAR-SoC-Knoten entsteht.

Den Kern eines einzelnen CAR-SoC bildet der CarCore-Prozessor. Die folgenden Abschnitte beschreiben kurz seinen Aufbau und Funktionsweise.

2.2.3.1 Prozessorstruktur

Bei dem CarCore-Prozessor (Abbildung 2.2) handelt es sich um einen zweifach superskalaren SMT-Prozessor [61, 36], welcher binärkompatibel zu Infineons TriCore-Architektur [17] ist. Innerhalb der einzelnen Stufen seiner Pipelines können sich Befehle unterschiedlicher Threads in der Ausführung befinden. Er ist in der Lage, den Pipelines jeden Takt bis zu zwei Instruktionen zuzuweisen. Dazu enthält der Prozessor zwei Pipelines jeweils mit den Stufen *Decode*, *Execute* und *Write Back*. Die Stufen im vorderen Teil (*Instruction Fetch* und *Schedule*) werden von beiden Pipelines gemeinsam genutzt. Instruktionen werden geordnet in die Pipeline gegeben. Falls ein Datenbefehl direkt von einem Adressbefehl gefolgt wird, werden beide parallel in die Pipelines gegeben. Ansonsten wird die andere Pipeline mit einem Befehl aus einem anderen Thread befüllt.

Die Ablaufsteuerung der Threads erfolgt in zwei Schritten, nämlich in der *Schedule*-Stufe innerhalb der Pipeline sowie in einem dedizierten *Thread Manager*. Die Pipeline selbst erlaubt die überlappende Ausführung von bis zu vier Threads. Dazu besitzt sie vier *Instruction Windows*, die geholte Befehle der Threads puffern und aus denen die Schedule-Stufe die Zuweisung vornimmt. Außerdem besitzt die

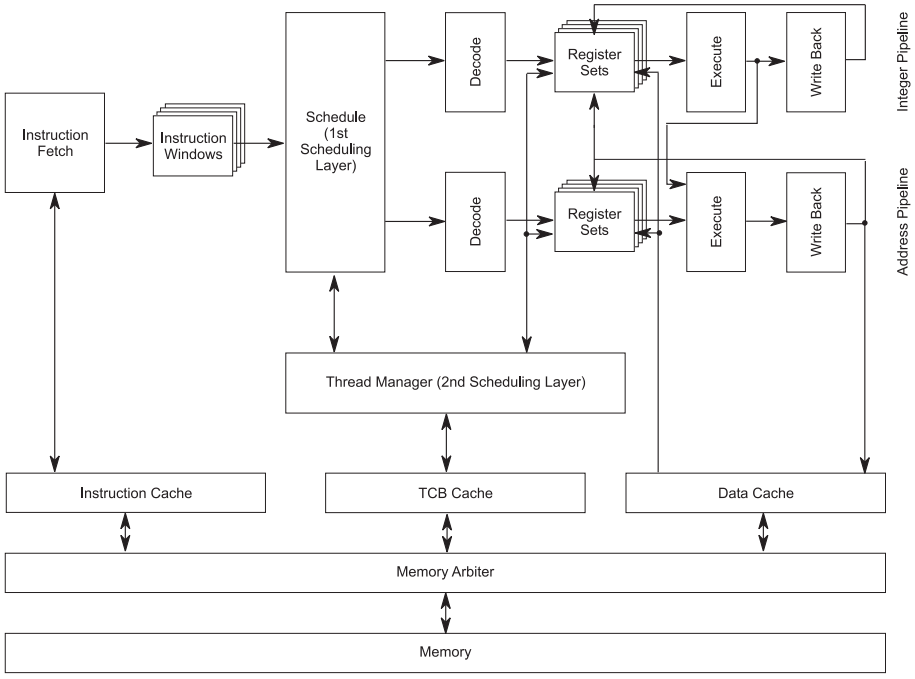


Abbildung 2.2: Architektur des CarCore-Prozessors

Pipeline je vier Adress- und Datenregistersätze, zwischen denen sie ohne Latenz umschalten kann.

Der *Thread Manager* erlaubt die Ausführung von prinzipiell unbegrenzt vielen Threads. Die Threads werden in einem eigenen Speicherbereich verwaltet, den sogenannten *Thread Control Blocks (TCBs)*. Über diesen Bereich erfolgt die Kommunikation mit der Threadverwaltung des Betriebssystems. Zur Ausführung werden die TCBs in Listen für die einzelnen Threadarten verwaltet, die vom Thread Manager gemäß dem Schedulingverfahren abgearbeitet werden. Er legt dabei fest, welche Thread-Kontexte in die Pipeline geladen und ausgeführt werden.

2.2.3.2 Thread Manager

Der Scheduler von CarCore verwendet ein zeitbasiertes Schedulingverfahren, bei dem die zur Verfügung stehende Rechenzeit in Perioden gleicher Länge unterteilt ist. Innerhalb einer solchen Periode können Threads mit den folgenden Schedulingstrategien ausgeführt werden:

Dominant Time Sharing Ein DTS-Thread [37] erhält für eine gegebene Anzahl Takte innerhalb einer Periode höchste Priorität (*Cycle Quantum*). Insbesondere werden seine Speicherzugriffe immer bevorzugt ausgeführt. Um dabei Kollisionen mit anderen DTS-Threads zu vermeiden, kann zu jeder Zeit

maximal ein DTS-Thread aktiv sein. DTS-Threads dienen somit als Container für Anwendungen mit **harten Echtzeitanforderungen**.

Periodic Instruction Quantum Für PIQ-Threads [38] versucht der Hardware-Scheduler, langfristig einen vorgegebenen Befehlsdurchsatz zu erreichen. Vorgegeben wird dabei die Anzahl der Befehle, die pro Periode ausgeführt werden sollen. Tatsächlich kann dieses *Instruction Quantum* aber nicht für einzelne Perioden garantiert werden. Durch Übertragung der nicht ausgeführten Befehle in die folgenden Perioden wird aber versucht, zumindest auf längere Sicht eine garantierte Rechenzeit zuzuteilen. Damit eignen sich PIQ-Threads zur Implementierung von Anwendungen mit **weichen Echtzeitanforderungen**.

Round Robin by Instruction Quantum Die verbleibende Rechenzeit innerhalb einer Periode wird an RRIQ-Threads vergeben. Diese werden untereinander gemäß eines vorgegebenen *Instruction Quantum* gewichtet. Allerdings wird für diese Art von Threads keinerlei Garantie bezüglich ihres Zeitverhaltens abgegeben. Damit eignen sie sich nur für die Implementierung von Anwendungen **ohne Echtzeitanforderungen**.

Mit diesen Schedulingverfahren bietet der CarCore die Möglichkeit, harte Echtzeitanwendungen parallel zu anderen Anwendungen auszuführen, ohne im Zeitverhalten von diesen beeinflusst zu werden.

2.2.3.3 Monitoring-Schnittstelle des Hardware-Schedulers

DTS-Threads dienen zur Ausführung von harten Echtzeitanwendungen. Die für DTS-Threads genutzte Rechenzeit kann durch Aufsummieren der Cycle-Quanta aller DTS-Threads berechnet werden. Bei PIQ-Threads führt diese Vorgehensweise über deren Instruction Quanta zu ungenauen Ergebnissen. Da diese Threads Latenzen anderer, parallel ablaufender Threads ausnutzen können, andererseits aber aufgrund der Abfolge ihrer Instruktionen unterschiedliche Latenztakte haben, kann die Anzahl der tatsächlich in einer Runde ausgeführten Befehle variieren.

Als eine Schnittstelle zum Betriebssystem besitzt der Hardware-Scheduler deshalb zwei spezielle Register, die eine genaue Überwachung des Laufzeitverhaltens aller PIQ-Threads ermöglichen:

- **EARLY_SATURATION** gibt die Anzahl der Takte vom Ende des letzten PIQ-Threads der Runde bis zum tatsächlichen Runden Ende an. Voraussetzung ist dabei, dass wirklich alle PIQ-Threads voll ausgeführt werden konnten.

- Konnte wenigstens einem PIQ-Thread nicht sein volles Instruction Quantum zugeteilt werden, so setzt der Scheduler **UNSAT_QUANTUM** auf die Differenz zwischen gewünschtem Instruction Quantum und tatsächlich zugeteilten Instruktionen. Die Quanta weiterer PIQ-Threads, die in der Runde gar nicht ausgeführt werden konnten, werden dabei nicht beachtet.

2.3 Autonomic und Organic Computing

Auch im Bereich von Serversystemen haben Entwickler und Nutzer eine steigende Komplexität zu bewältigen. Aus diesem Grund stellte IBM im Jahr 2001 die Konzepte des *Autonomic Computing* vor [16]. Kephart und Chess haben die Anforderungen an autonome Computersystem später konkretisiert [20]. So sollen zukünftige Computersysteme die folgenden Eigenschaften besitzen:

- **Selbstkonfigurierend** (*Self-Configuring*): Die Einführung neuer Komponenten in ein bestehendes komplexes System ist ein aufwändiger und fehleranfälliger Prozess. Zukünftige autonome Systeme führen die dabei notwendigen Konfigurationsarbeiten an den neuen Komponenten sowie die Anpassung des Systems selbstständig durch.
- **Selbstheilend** (*Self-Healing*): In komplexen Systemen ist die Fehleranalyse sehr aufwändig. Mit Hilfe neuer Techniken sollen zukünftige Computersysteme Fehler selbstständig erkennen und automatisch beseitigen.
- **Selbstoptimierend** (*Self-Optimizing*): Die Arbeitsweise des Systems wird kontinuierlich verbessert. Dazu werden die vielen einstellbaren Parameter, die ein solches System gewöhnlich bietet, zur Laufzeit überwacht und auf Verbesserungsmöglichkeiten untersucht. Falls solche Verbesserungsmöglichkeiten gefunden werden, so werden diese selbstständig durch Umkonfigurieren des Systems sowie eventuelle Verlagerung von Diensten auf andere Knoten aktiviert.
- **Selbstschützend** (*Self-Protecting*): Zwar stellt die heutige Technik bereits Konzepte wie Firewalls zum Schutz von Systemen bereit, die Entscheidung über den Einsatz solcher Konzepte muss aber von Menschen getroffen werden. Dabei werden auch immer nur bestimmte Teile bzw. Schnittstellen eines Systems geschützt. Autonomic-Computing-Systeme können darüber hinaus das Gesamtsystem umfassend schützen. Außerdem können sie Probleme vorhersagen und frühzeitig Maßnahmen zum Schutz davor ergreifen.

Diese Eigenschaften werden zusammenfassend als *Self-CHOP-Properties* oder auch *Selbst-X-Eigenschaften* bezeichnet.

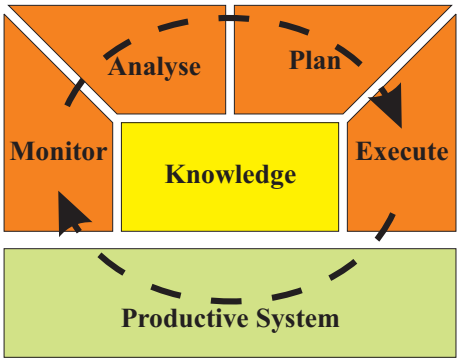


Abbildung 2.3: Die MAPE-Architektur nach [20]

Als Leitfaden für die die Implementierung der Self-CHOP-Eigenschaften wurde die *MAPE-Architektur* (Abbildung 2.3) vorgeschlagen. Dabei wird ein *Produktivsystem* (Managed Element) um einen *Autonomic Manager* erweitert. Diese beiden zusammen bilden eine *Autonome Komponente* (Autonomic Element). Der Autonomic Manager selbst ist wiederum aus mehreren Komponenten aufgebaut. Die Elemente *Monitor*, *Analyze*, *Plan* und *Execute* bilden eine Rückkopplungsschleife über dem Produktivsystem und können auf ein gemeinsames *Wissen* (Knowledge) zugreifen. Diese Einzelkomponenten stellen die folgenden Funktionalitäten zur Verfügung:

- Der **Monitor** überwacht das Systemverhalten anhand einzelner Überwachungspunkte.
- Die **Analyze**-Komponente analysiert die Rohdaten des Monitors und leitet Aussagen über das Systemverhalten her.
- Aus diesen Aussagen berechnet die **Plan**-Komponente einen Plan zur Verbesserung des Systemverhaltens.
- Die **Execute**-Stufe führt diesen Plan aus.
- Das Wissen (**Knowledge**) über das Produktivsystem beeinflusst das Verhalten der genannten Komponenten.

Einige Jahre später wurden diese Ideen dann auch für eingebettete Systeme unter dem Begriff des *Organic Computing* (OC) aufgegriffen [39]. Das Forschungsfeld wurde dabei um den Bereich des *emergenten Verhaltens* erweitert. Anwendungen hierzu werden oft durch Muster in der Natur inspiriert. Typische Vorbilder sind hierbei Schwarminsekten wie Ameisen oder Bienen. Jedes einzelne Individuum eines solchen Schwarmes verfügt nur über ein sehr begrenztes Wissen und geringe Aktionsmöglichkeiten. Durch die Interaktion dieser Individuen können aber komplexe Systeme entstehen. Dieses Verhalten ist insofern emergent, als dass es erst durch die Interaktion zu Tage tritt. Es kann aber nicht aus den Verhaltensweisen

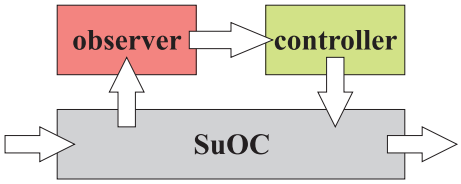
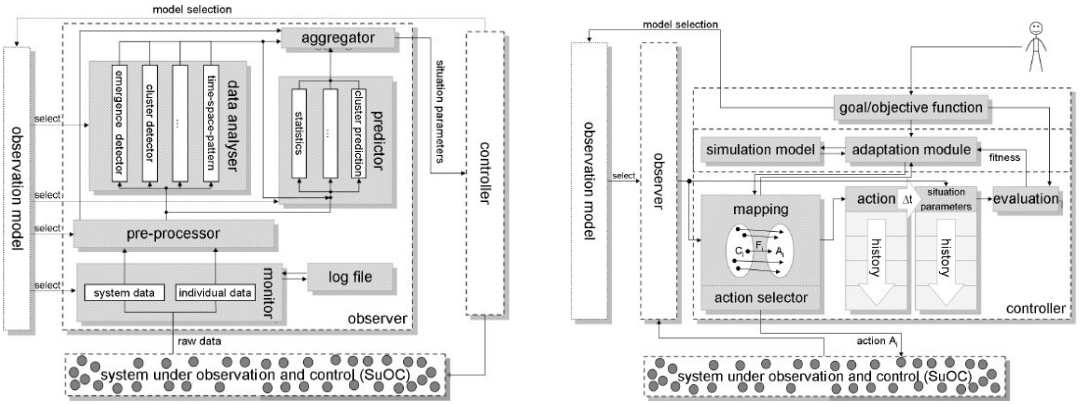


Abbildung 2.4: Observer/Controller-Grundarchitektur



(a) Oberver (b) Controller

Abbildung 2.5: Generische Observer/Controller-Architektur (aus [50])

eines isolierten Individuums hergeleitet werden. Ein wichtiger Forschungsaspekt hierbei ist es, dieses emergente Verhalten gezielt zu kontrollieren.

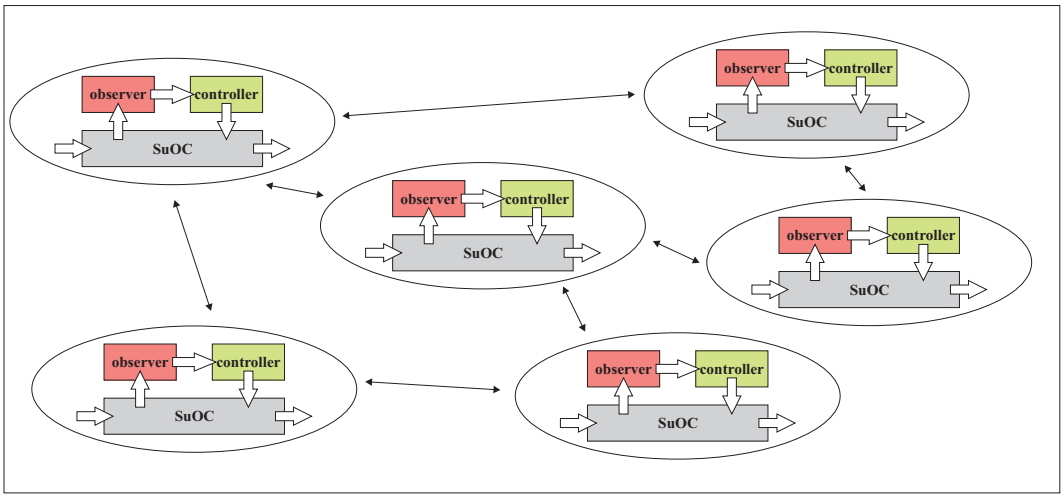
Richter et al. [50] stellten dazu in der Folgezeit eine Observer/Controller-Architektur als Entwurfparadigma für OC-System auf. Die Grundidee ähnelt der MAPE-Architektur des Autonomic Computing. Ein Produktivsystem (*System under Observation/Control, SuOC*) ist in eine Rückkopplungsschleife aus Observer und Controller eingebettet (Abbildung 2.4). Das Produktivsystem selbst ist bereits ohne diese Einbettung voll funktionsfähig. Die überlagerten Observer/Controller-Komponenten verbessern den Betrieb des Produktivsystems etwa im Hinblick auf Ausfallsicherheit oder Anpassungsfähigkeit. Der Observer sammelt hierzu Informationen über das Produktivsystem, aggregiert sie und leitet sie an den Controller weiter. Dieser bewertet die Daten mithilfe von Zielfunktionen und wählt darauf basierend Aktionen aus, mit deren Hilfe er das SuOC in die gewünschte Richtung lenkt.

Der Observer setzt sich aus mehreren Komponenten zusammen (Abbildung 2.5(a)). Ein *Monitor* überwacht relevante Attribute des SuOC und erzeugt aus diesen Daten eine Zeitreihe. Die Überwachungsfrequenz wird dabei durch das *Observation Model* festgelegt. Diese Daten speichert er in einer *Logdatei*, die durch den Prädiktor oder für eine weiterführende Analyse verwendet wer-

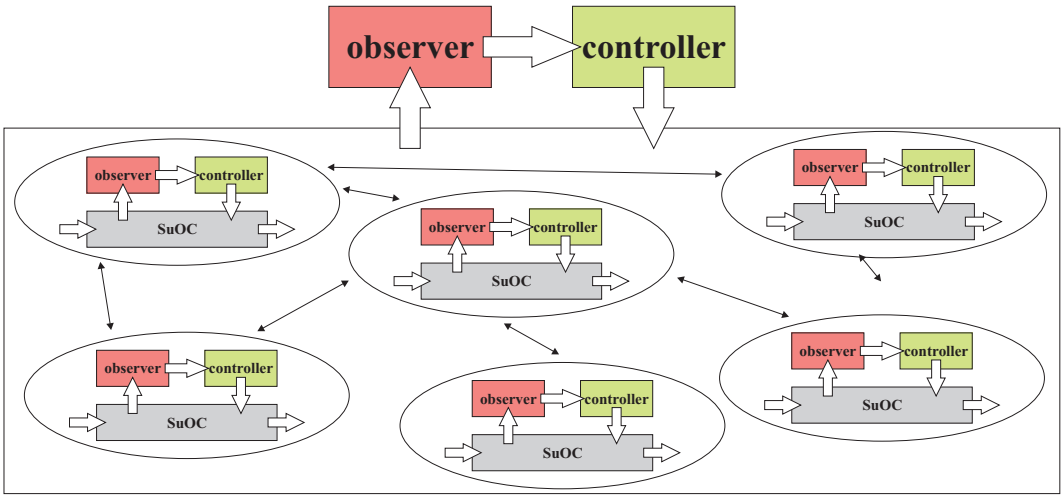
den kann. Ein *Präprozessor* leitet aus den Rohdaten weitere Informationen ab. Er entscheidet auch, welche Daten für die weitere Verarbeitung relevant sind und leitet diese an die Datenanalyse und den Prädiktor weiter. Die *Analyseeinheit* kann nun verschiedene Analysemethoden auf diese Daten anwenden, etwa Clusteranalyse oder Emergenzerkennung. Das Ergebnis dieser Berechung ist eine systemweite Beschreibung des aktuellen Zustands. Der *Prädiktor* verarbeitet die vom Präprozessor und der Analyseeinheit kommenden Daten. Sein Ziel ist es, zukünftige Systemzustände vorherzusagen, insbesondere um unerwünschtes Verhalten zu vermeiden. Diese Vorhersagen basieren auf einer Analyse früherer Systemzustände, die der Prädiktor für ein gegebenes Zeitfenster speichert. Der *Aggregator* sammelt die Ergebnisse der Analyseeinheit, des Prädiktors und gegebenenfalls auch Rohdaten aus dem Präprozessor. Auch er speichert lokal die Informationen vergangener Zustände. Diese nutzt er, um den aktuellen Datensatz Filteroperationen zu unterziehen, etwa um Rausch-Effekte zu eliminieren. Die aktuellen und früheren Werte beschreiben den aktuellen Systemzustand sowie die Dynamik des SuOC. Der Aggregator leitet diese Informationen an den Controller weiter.

Auch der Controller besteht aus mehreren Komponenten (Abbildung 2.5(b)). Einen direkten Schluss der Regelschleife stellt der *Action Selector* dar. Er bildet die aus dem Observer empfangenen Daten zur Systemsituation auf die am besten geeignete Aktion ab und leitet diese an das SuOC weiter. Dies ermöglicht eine schnelle Reaktion auch unter Echtzeitbedingungen. Zusätzlich sammelt der Controller Informationen über ausgewählte Aktionen zu einem Zeitpunkt t sowie die Systemzustände zu Zeitpunkten $t + \Delta t$ (Δt fest vorgegeben), um die Auswirkungen der gewählten Aktionen festzuhalten. Diese evaluiert er mithilfe von Techniken des Machine Learning, um für die Aktionen neue Fitness-Werte zu berechnen. Das Adaptionsmodell passt auf Basis dieser Fitness-Werte die Zustand-Aktion-Abbildungen des Action Selectors an. Diese Adaption kann zur Laufzeit geschehen und durch einen modellbasierten Lernprozess unter Nutzung des *Simulation Model* unterstützt werden. Das gesamte Verhalten des Controllers wird durch *Zielfunktionen* gesteuert, die durch den Benutzer oder Entwickler vorgegeben sind. Sie beeinflussen sowohl das Adaptionsmodul als auch die Wahl des Observation Model.

Im einfachsten Fall wird das Gesamtsystem mit einer solchen Observer/Controller-Komponente versehen (Abbildung 2.4). Moderne Computersysteme bestehen allerdings typischerweise aus mehreren Komponenten. Jede Einzelkomponente wird hier durch eine Observer/Controller-Komponente überwacht und geregelt. Diese Kontrolleinheiten können wiederum untereinander Informationen austauschen. Dadurch ergibt sich ein verteiltes System wie in Abbildung 2.6(a). Dieses kann noch um eine globale Observer/Controller-



(a) Verteilte Observer/Controller-Architektur



(b) Verteilte Observer/Controller-Architektur mit übergeordneter Regelschleife

Abbildung 2.6: Modellarchitekturen mit der Observer/Controller-Architektur

Komponente erweitert werden (Abbildung 2.6(b)), die das Verhalten des Gesamtsystems kontrolliert.

2.3.1 Anforderungen an das Betriebssystem

Durch die Einführung der Konzepte des Autonomic/Organic Computing in den Bereich eingebetteter Echtzeitsysteme ergeben sich weitere Anforderungen. Zur Realisierung von AC-/OC-Funktionalitäten sind zwei Implementierungsansätze denkbar. Einerseits kann die Funktionalität direkt in die Anwendung integriert werden, sofern sie echtzeitfähig ist. Sollte dies nicht der Fall sein, so kann sie als separater Dienst in Form eines Helper Threads implementiert werden. Hierzu muss zunächst Anforderung RTOS-2 (Echtzeitscheduling) aus Abschnitt 2.1.1 unter Nutzung von Anforderung RTOS-1 (Mehrfädigkeit) wie folgt erweitert werden:

RTOS-2' (Helper Threads) Das Betriebssystem erlaubt es, zusätzliche Anwendungen im völliger zeitlicher Isolation von den harten Echtzeitthreads laufen zu lassen. Es muss also möglich sein, parallel zu den Anwendungsthreads weitere **Helper Threads** laufen zu lassen, ohne dass sich das Zeitverhalten der Anwendungsthreads ändert. Wie in späteren Kapitel gezeigt wird, lassen sich viele Selbstadaptionsmechanismen zwar direkt in Betriebssystemdienste integrieren, dies führt aber bei einer WCET-Analyse zwangsläufig zu höheren Laufzeitabschätzungen. Gerade bei aufwändigeren Techniken ist es deshalb von Vorteil, wenn diese separat laufen und nur geringen Einfluss auf die WCET einer Anwendung haben.

Kernpunkte jeder AC-/OC-Architektur sind zum einen die umfassende Überwachung des Systemzustands, und zum anderen entsprechende Eingriffsmöglichkeiten. Daraus ergeben sich folgende weitere Anforderungen:

RTOS-5 (Monitoring) Das Betriebssystem stellt umfassende, feinkörnige **Laufzeitinformationen** über Systemparameter und laufende Threads zur Verfügung. Aus diesen Laufzeitinformationen kann ein genaues Bild über den aktuellen Systemzustand abgeleitet werden.

RTOS-6 (Reaktionen) Entsprechend müssen ebenso feinkörnige **Eingriffsmöglichkeiten** vorhanden sein, um das Laufzeitverhalten des Systems gezielt zu beeinflussen. Die Art dieser Eingriffsmöglichkeiten ist stark von deren Ort abhängig. Diese Eingriffsmöglichkeiten sowie die oben genannten Laufzeitinformationen sind Grundvoraussetzung zur Implementierung der MAPE- oder Observer/Controller-Architektur auf einem Produktivsystem.

RTOS-7 (Codemigration) Ein Konzept für **mobilen Code** ermöglicht das Verschieben von Anwendungen zwischen Knoten in einem verteilten System. So

kann etwa beim drohendem Ausfall einzelner Knoten die weitere Verfügbarkeit deren Dienste sichergestellt werden, indem diese auf andere, voll funktionsfähige Knoten verlagert werden.

RTOS-8 (Sicherheit) Geeignete **Sicherheitsmaßnahmen** stellen die Funktionsfähigkeit des Gesamtsystems sicher. Diese Anforderung betrifft mehrere Aspekte. So gilt es, Funktionsstörungen durch fehlerhafte Programme zu vermeiden, indem diese möglichst isoliert ausgeführt werden. So können diese keine anderen parallel laufenden Anwendungen beeinflussen, wodurch die Zuverlässigkeit des Systems sichergestellt wird. Zudem muss das System auch gegen gezielte Angriffe etwa durch Schadsoftware geschützt werden.

All diese Anforderungen beziehen sich zunächst auf den inneren Aufbau des Betriebssystems. Gleichzeitig ist es notwendig, entsprechende Werkzeuge bereitzustellen, die die Entwicklung und Nutzung der geforderten Funktionalitäten auf Anwendungsebene unterstützen.

Die Funktionalitäten **RTOS-5/Monitoring** und **RTOS-6/Reaktionen** werden idealerweise anhand einer einheitlichen Schnittstelle bereitgestellt. Das Betriebssystem selbst kann zunächst nur Laufzeitinformationen über seine eigenen Komponenten liefern. Ebenso verhält es sich mit den Eingriffsmöglichkeiten. Deshalb ist es wichtig, von allen zusätzlichen Komponenten wie Anwendungen und Treibern die Implementierung einer solchen Schnittstelle zu fordern. Diese Schnittstelle ist wiederum möglichst allgemein zu halten, um eine breite Einsetzbarkeit zu gewährleisten.

Die Grundvoraussetzung für **RTOS-7/Codemigration** ist das Vorhandensein eines *dynamischen Linkers* oder *Runtime Linkers*. Damit kann neuer Anwendungscode zur Laufzeit in ein System eingebunden werden. Diese Funktionalität ist auf Betriebssystemen, die die POSIX-Spezifikation [47] implementieren, normalerweise vorhanden. Für die volle Funktionsfähigkeit ist es zudem nötig, den Zustand einer Anwendung speichern zu können, um diesen zusammen mit dem Code auf den Zielknoten zu migrieren.

Heutige Betriebssysteme für eingebettete Systeme stellen üblicherweise schon eine Reihe von Sicherheitsmaßnahmen (**RTOS-8/Sicherheit**) zur Verfügung. Diese sind unter anderem die Nutzung von hardware-basierten Speicherschutztechniken, oder, auf POSIX-basierten Systemen, der Einsatz einer Rechteverwaltung.

Abbildung 2.7 fasst die genannten Anforderung zusammen und ordnet sie den Bereichen *Echtzeit* und *Autonomic/Organic Computing* zu. Auf dieser Basis soll nun eine Betriebssystemarchitektur entworfen werden, welche all diese Anforderungen erfüllt.

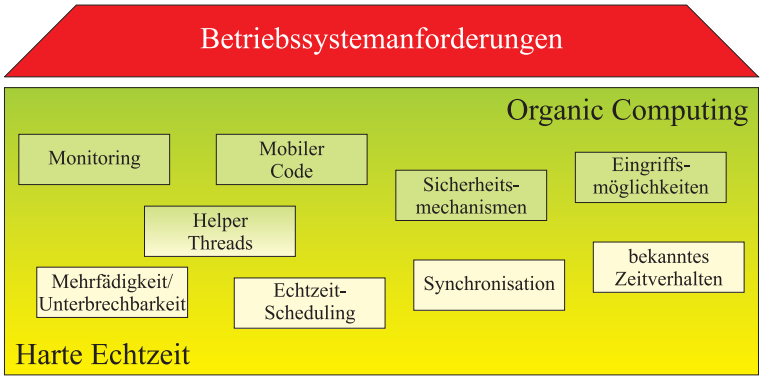


Abbildung 2.7: Anforderungen an ein organisches Echtzeitbetriebssystem

2.3.2 Anforderungen an das Organic Management

Auch für ein Organic Management ergeben sich durch seinen Einsatz innerhalb von eingebetteten Echtzeitsystemen gewisse Anforderungen.

OM-1 (Echtzeitfähigkeit) Die Teile des Organic Managements, die direkt mit Anwendungen interagieren, dürfen deren Zeitverhalten nicht unvorhersehbar beeinflussen. Bei einer Integration in Echtzeitanwendungen müssen sie auch selbst echtzeitfähig sein.

OM-2 (Isolierbarkeit) Nicht echtzeitfähige Teile des Organic Managements müssen sich in zeitlicher Isolation zu den Anwendungen des Knotens ausführen lassen. Sie müssen sich also in Form eines Helper Threads implementieren lassen.

OM-3 (Effiziente Algorithmen) Auch wenn das Management durch Anforderung OM-2 zeitlich von den Anwendungen isoliert ist, steht ihm nur begrenzte Rechenzeit zur Verfügung. Insofern müssen die Management-Algorithmen sehr laufzeiteffizient sein, um eine schnelle Reaktion zu ermöglichen.

OM-4 (Nicht-negative Wirkung) In harten Echtzeitsystemen kann ein Fehlverhalten des Systems katastrophale Folgen haben. Es muss daher sichergestellt werden, dass das Organic Management nur solche Reaktionen auswählt, die den aktuellen Systemzustand nicht verschlechtern. In Bezug auf Anwendungen ohne Echtzeitanforderungen kann ein Teilausfall der Funktionalitäten (*graceful degradation*) hingenommen werden, solange dadurch die Kernfunktionaliät aufrechterhalten werden kann.

Anforderung OM-1 hinsichtlich der Echtzeitfähigkeit berücksichtigt die generische Observer/Controller-Architektur durch ihren Action Selector. Sie verzichtet aber darauf, dies auch von den anderen Komponenten des Regelkreises zu fordern.

Ebenso ist Anforderung OM-4 schon definiert, wird aber durch den möglichen Einsatz von Lerntechniken relativiert.

2.3.3 Entscheidungstechniken für OC-Systeme

Dieser Abschnitt stellt einige Konzepte vor, die in der Controller-Komponente von OC-Systemen zu Aktionsauswahl und auch Weiterentwicklung der Aktionsbasis genutzt werden.

2.3.3.1 Learning Classifier Systems

Learning Classifier Systems (LCS, [15]) bestehen aus einem Regelsatz, der laufend mit Hilfe eines genetischen Algorithmus' und einer Lernfunktionen optimiert wird. Der Regelsatz selbst basiert im Allgemeinen auf einfachen Zeichenketten aus *wahr/falsch/egal*-Werten (Symbole `1/0/#`) für die Eingabe bzw. *wahr/falsch*-Werten für die Ausgabe. Jede Regel ist zusätzlich mit einem oder mehreren Attributen ausgestattet, welche von der Lernfunktion und dem genetischen Algorithmus zur Weiterentwicklung der Regelpopulation benutzt werden. Die Art und Bedeutung dieser Attribute hängt dabei von der Art des eingesetzten LCS ab. In jedem Klassifizierungzyklus wählt das LCS basierend auf den Eingabewerten eine geeignete Regel („Classifier“) und führt die entsprechende Aktion aus. Im Idealfall bewegt sich das System damit einen Schritt auf einen Zielzustand zu. Eine Bewertungsfunktion bewertet den Erfolg der ausgewählten Aktion. Mithilfe dieser Bewertung verändert das LCS die Attribute der vorher ausgewählten Regeln. Der genetische Algorithmus führt durch Mutation und Rekombination bestehender Regeln neue Regeln in den Regelsatz (*Population*) ein. Alte Regeln entfernt das LCS im Lauf der Zeit aus der Population. Wichtige Kriterien hierfür sind die Attributswerte der Regeln, die das LCS mithilfe der Lernfunktion stetig weiterentwickelt. Diese geben an, wie sich eine Regel in der Vergangenheit bewährt hat. Somit werden vor allem Regeln, die dem System wenig Nutzen brachten, aus der Population entfernt.

Der Regelsatz eines LCS stellt also die Aktionsbasis für den Action Selector dar. Die Bewertungsfunktion dient der Evaluierung, während der genetische Algorithmus das Adaptionsmodul des Controllers repräsentiert. Im Bereich des Organic Computing kommen Learning Classifier Systems unter anderem bei Schöler und Müller-Schloer [54] zum Einsatz.

Ein Vorteil von LCS ist die beschränkte Länge des Parametersatzes. Somit lässt sich die Datenbasis eines LCS sehr platzsparend implementieren. Ebenso können Regeln und Zustände sehr effizient miteinander verglichen werden. Solange der Umfang des Regelsatzes ausreichend beschränkt ist, kann man auch eine ange-

messen beschränkte Laufzeit der Klassifizierung garantieren. Nachteilig für eingebettete Echzeitsysteme ist allerdings die zufallsorientierte Arbeitsweise des genetischen Algorithmus't. Dadurch können Regeln entstehen, die das Systemverhalten negativ beeinflussen. Gerade in *harten* Echtzeitsystem muss dies aber strikt vermieden werden. LCS erlauben nur eine schrittweise Änderung des Systemzustands, da in jedem Klassifizierungszyklus nur eine Aktion ausgeführt wird.

2.3.3.2 Automatisierte Planer

Im Gegensatz zu LCS sind automatisierte Planer [51, 13] auch in der Lage, ganze Aktionenfolgen herzuleiten, die ein System aus einem unerwünschten Ausgangszustand in mehreren Einzelschritten in den gewünschten Zielzustand überführen. Die zur Verfügung stehenden Aktionen mit ihren Vor- und Nachbedingungen werden üblicherweise in entsprechenden Planersprachen wie etwa PDDL angegeben (*Planning Domain Definition Language*). Im Zusammenhang mit dem aktuellen Zustand spannen die Aktionen einen Graphen auf. Die Knoten des Graphen stellen dabei mögliche Folgezustände dar. Die Kanten repräsentieren die Aktionen, über die diese Zustände erreicht werden können. Aufgabe des Planers ist es nun, für einen gegebenen Startzustand einen Weg durch diesen Graphen zu dem gewünschten Zielzustand zu finden. Durch den Nutzer vorgegebene Nebenbedingungen können diese Wege zusätzlich einschränken.

Für die Aktionen werden Voraussetzung und Effekt meist in Form logischer Formeln angegeben. Durch Kombination dieser Informationen mit dem aktuellen Systemzustand kann dieser graduell geändert werden, bis der Zielzustand erreicht ist. Durch Nebenbedingungen kann außerdem sichergestellt werden, dass schlechte Zustände gemieden werden. Vorteilhaft ist auch das offene Konzept, das es ermöglicht, auch zur Laufzeit neue Aktionen einzufügen. Allerdings sind die Planungsprozesse im Allgemeinen sehr aufwändig, womit ein direkter Einsatz in leistungsbeschränkten eingebetteten Systemen stark beschränkt wird. Im Bereich des Organic Computing wird ein automatisiertes Planungsverfahren für vertrauenswürdige selbstorganisierende Systeme eingesetzt [52].

Der Planungsalgorithmus selbst ist über die zugehörige Beschreibungssprache von der Problemdomäne entkoppelt und damit von dieser unabhängig. Insofern sind auch Planungsalgorithmen *generisch*. Problematisch ist allerdings der hohe Rechenaufwand, der für eine Planerstellung benötigt wird. Für den Einsatz in ressourcenarmen eingebetteten Echtzeitsystemen erscheinen Planer daher wenig geeignet. Dies gilt umso mehr, wenn sie nur einen geringen Teil der Rechenleistung des Systems in Anspruch nehmen dürfen.

2.4 Aktuelle Echtzeitbetriebssysteme und Echtzeit-Middleware

2.4.1 Choices - Ein selbstheilendes Betriebssystem

Choices (*Class Hierarchical Open Interface for Custom Embedded Systems*) wurde von Campbell et al. [9] ursprünglich als Betriebssystem für große Multiprozessorsysteme entworfen. Die einzelnen Knoten in solchen Systemen sind üblicherweise über einen gemeinsamen Speicher oder eine schnelle Vernetzung untereinander verbunden. Der Entwurf von Choices zielt aber auf ein deutlich größeres Anwendungsfeld ab. So soll Choices auch im Bereich eingebetteter Echtzeitsysteme zum Einsatz kommen, wie sie im Bereich der Luftfahrt zu finden sind. Dies betrifft nicht nur Steuerung an Bord eines Flugzeugs, sondern auch die Rechensysteme in Flugverkehrskontrollstellen und Flugleitsystemen.

Um den Anforderungen aus den sehr verschiedenen Anwendungsbereichen gerecht zu werden, wurde Choices als stark anpassbares Betriebssystem entwickelt. Kernfunktionen werden innerhalb einer Klassenhierarchie abgebildet. Einzelne Klassen in dieser Hierarchie können problemspezifisch ausgewählt und kombiniert werden. Dadurch ist es möglich, ein auf die Bedürfnisse einer speziellen Anwendung zugeschnittenes Betriebssystem bereitzustellen.

David et al. [12] nutzen Choices als Grundlage, um ein selbstheilendes Betriebssystem zu entwickeln. Dabei wird ein problemorientierter Ansatz verfolgt. Für verschiedene Arten von Fehlermodellen werden entsprechende Korrekturmechanismen bereitgestellt.

Grundlage für alle Mechanismen bildet zum einen eine Ausnahmebehandlung, die das Betriebssystem durch Unterstützung von C++-Ausnahmen bereitstellt. Außerdem können Prozessorausnahmen in entsprechende C++-Objekte abgebildet werden. Damit werden alle Ausnahmesituationen auf eine einheitliche Schnittstelle abgebildet. Zum anderen setzt Choices auf die Isolation der einzelnen Softwarekomponenten. Dadurch können sich Fehler nicht über die betroffene Komponente hinaus fortpflanzen. Entsprechend können Selbstheilungsmaßnahmen auf die einzelne Komponente beschränkt werden. Diese Isolation ist in Choices mithilfe von virtuellem Speicher und Speicherschutztechniken implementiert. Einzelne Komponenten werden dabei von Klassen gekapselt. Diese Wrapper besitzen über die Klassenhierarchie von Choices die Fähigkeit, all jene Ausnahmen zu behandeln, die von den Komponenten nicht gefangen werden werden können.

In Choices sind verschiedene Techniken zur Fehlererkennung und Systemwiederherstellung implementiert:

- *Code reloading:* Falls Speicherinhalte durch eine fehlerhafte Programmausführung beschädigt werden, so werden die Daten aus einem nichtflüchtigen Speicher erneut geladen. Beschädigte Speicherinhalte werden erkannt, wenn der Prozessor eine Ausnahme aufgrund eines ungültigen Befehls auslöst. Zusätzlich können auch regelmäßige Codeprüfungen genutzt werden, um Speicherfehler bereits zu erkennen, bevor sie weitere Fehler auslösen.

- *Component micro-rebooting:* Durch einen Micro-Reboot wird eine Komponente neu initialisiert. Danach wird versucht, den Dienst der Komponente erneut in Anspruch zu nehmen. Auf diesem Weg können Fehler in Datenstrukturen innerhalb des Betriebssystemkerns behoben werden.

- *Automatic service restarts:* Bestimmte Dienste eines Betriebssystems sind essentiell für die Funktionalität des Systems. Falls der Ausfall eines solchen Dienstes festgestellt wird, so wird durch seinen automatischen Neustart die volle Funktionsfähigkeit des Systems sichergestellt. Choices stellt auch hier eine entsprechende Kapselung zur Verfügung. Außerdem nutzt Choices *lock-tracking*, um alle Sperren verfolgen zu können, die eine Komponente belegt. Dies ist notwendig, um diese im Falle eines Neustarts der Komponente auch freigeben zu können.

- *Watchdog-based recovery:* Watchdog-Timer werden eingesetzt, um sicherzustellen, dass das Betriebssystem in seiner Arbeit fortschreitet. Falls der Timer nicht regelmäßig durch das Betriebssystem zurückgesetzt wird, so wird er bei seinem Ablaufen den Prozessor benachrichtigen. Dadurch wird üblicherweise ein kompletter Neustart des Systems ausgelöst. Durch geeignete Modifikation des Watchdog-Mechanismus' wird verhindert, dass der Inhalt das Arbeitsspeichers verloren geht. Ebenso bleibt der Zustand der MMU erhalten. Damit kann nach einer Neuinitialisierung der reguläre Betrieb fortgesetzt werden. Einzig der Zustand des letzten Prozesses, der lief, als der Watchdog-Timer auslöste, geht verloren.

- *Transactional components:* Die Systemkomponenten werden um ein Transaktionsmodell erweitert. Fehleranfällige Operationen werden durch Transaktionen gekapselt. Falls eine solche Transaktion fehlschlägt, so kann der vorhergehende Zustand der Komponente wiederhergestellt werden. Transaktionen bieten somit im Vergleich zu dem oben genannten *micro-rebooting* feinere Eingriffsmöglichkeiten. Durch den Transaktionsmechanismus kann sichergestellt werden, dass wichtige Statusinformationen erhalten bleiben, welche durch einen Micro-Reboot verlorengehen würden.

- *Process-level recovery:* Zuletzt ist es möglich, den kompletten Zustand einer Anwendung zu sichern. Dieser kann dann nach einem Neustart des Systems gezielt wiederhergestellt werden. Betriebssysteminterne Zustände hingegen

werden durch den Neustart neu initialisiert, wodurch mögliche transiente Fehler beseitigt werden.

Die genannten Funktionalitäten wurden für den OMAP1610 H2 von Texas Instruments implementiert. Dabei handelt es sich um einen Prozessor für den Einsatz in Mobiltelefonen. Weitere Evaluierungen wurden an ARM Integrator unter Zuhilfenahme von QEMU durchgeführt. Damit wurde die Praktikabilität der vorgestellten Techniken für den Einsatz in eingebetteten Systemen unter Beweis gestellt.

Ohne Beachtung bleiben bei dieser Arbeit allerdings die besonderen Anforderungen von harten Echtzeitanwendungen.

2.4.2 QNX

QNX [48] entstand 1980 als Echtzeitbetriebssystem mit Mikrokernel. Als sich der POSIX-Standard [47] immer weiter durchsetzte, wurde der Kernel neu implementiert. Dadurch wurde eine vollständige Kompatibilität zu POSIX und SMP (*Symmetric MultiProcessing*) erreicht. Seit 2001 wird das Ergebnis dieser Arbeit unter der Bezeichnung *QNX Neutrino* vermarktet. Seit 2007 ist der Quellcode des QNX-Neutrino-Kernels für die nicht-kommerzielle Nutzung öffentlich verfügbar.

Der Betriebssystemkern besteht nur aus wenigen Komponenten. Dabei handelt es sich um den Prozessor-Scheduler, Mechanismen zur Interprozesskommunikation, die Unterbrechungsbehandlung sowie verschiedene Zeitgeber. Alle weiteren Dienste werden als Nutzerprozesse mit eingeschränkten Rechten ausgeführt. Dies betrifft auch Gerätetreiber.

Die Kommunikation zwischen zwei Nutzerprozessen findet synchron statt. Die Interprozesskommunikation ist dabei eng an das Prozessor-Scheduling gekoppelt. Dadurch sind diese Kommunikationsmechanismen sehr universell einsetzbar. Durch seine Mikrokern-Architektur kann QNX Neutrino auch als verteiltes Betriebssystem verwendet werden. Die dazu verwendeten Techniken werden als *Transparent Distributed Processing* bezeichnet. Insbesondere handelt es sich hierbei um ein Protokollmodul, mit dessen Hilfe mehrere Mikrokerne innerhalb eines Netzes verbunden werden. Dadurch werden Zugriffe auf entfernte Dienste für die Applikationssoftware transparent gemacht.

Eine weitere Besonderheit von QNX Neutrino ist der *Boot Loader*. Dieser kann ein komplettes System bestehend aus Betriebssystemkern, Gerätetreibern, Benutzerprogrammen und -bibliotheken aus einem einzigen Abbild laden. Dies ist insbesondere für den Entwurf eingebetteter Systeme hilfreich, da hier die komplette Abbilddatei im ROM-Speicher abgelegt werden (z.B. Flash) kann, aber keine Festplatten oder vergleichbare Techniken zur Verfügung stehen.

Der Einsatz von QNX Neutrino in symmetrischen Multiprozessorsytemen (SMP) wird unterstützt. Mit *Bound MultiProcessing* (BMP) stellt QNX Neutrino eine spezielle Schedulingtechnik bereit, die es erlaubt, ausgewählte Tasks an bestimmte Prozessoren zu binden. Als weitere Schedulingtechniken für einzelne Threads werden FIFO, Server Sporadic und Round Robin angeboten.

Durch *Adaptive Partitionierung* wird die Echtzeitfähigkeit und Sicherheit des Betriebssystems erhöht. Dabei wird einer Gruppe von Anwendungen eine minimale Prozessorleistung garantiert. Dies ist unabhängig von der Systemlast und wird selbst dann erfüllt, wenn gleichzeitig höherpriore Threads aktiv sind. Mit adaptiver Partitionierung können also Threads unter strikten Echtzeitbedingungen ablaufen. Gleichzeitig werden die einzelnen Gruppen weitgehend voneinander isoliert. Dadurch werden Beeinflussungen zwischen den Gruppen minimiert.

QNX Neutrino bietet einen einfachen Selbstheilungsmechanismus: Durch die Mikrokernarchitektur werden Anwendungen effizient gekapselt. Jede Anwendung läuft als *Server* mit einem eigenen Speicherbereich. Wird versucht auf den Speicher anderer Anwendung zuzugreifen, kann dies durch den Mikrokern unterbunden werden. Damit können Fehler schnell lokalisiert werden. Gleichzeitig wird verhindert, dass Fehler das restliche System negativ beeinflussen.

QNX Neutrino ist für folgende Prozessorarchitekturen verfügbar: x86, MIPS, PowerPC, SH-4, ARM, StrongARM und xScale.

2.4.3 VxWorks

VxWorks [65] ist ein proprietäres Echtzeitbetriebssystem, das von der Firma *Wind River Systems* entwickelt und vertrieben wird. Es wird hauptsächlich in der Luft- und Raumfahrt eingesetzt. Aber auch Automobilhersteller und -zulieferer nutzen es, so etwa Siemens VDO oder BMW.

VxWorks beinhaltet einen Multitasking-Kern, der sowohl präemptives als auch Round-Robin-Scheduling bietet. Optional ist auch eine Unterstützung für symmetrische Multiprozessorsysteme verfügbar. Ähnlich dem Bound Multiprocessing in QNX Neutrino besteht auch hier die Möglichkeit, eine Applikation an eine bestimmte CPU zu binden.

VxWorks bietet eine verbindungsorientierte Interprozesskommunikation. Diese unterstützt das TIPC-Protokoll (*Transparent Inter-Process Communication*), das eine ortstransparente Interprozesskommunikation innerhalb eines Rechen-Clusters ermöglicht. Da diese Kommunikation auf einem offenen Standard basiert, können damit auch Dienste angesprochen werden, die unter anderen Betriebssystemen wie etwa Linux laufen.

Durch den Einsatz von Speicherschutzmechanismen werden Anwendungsprogramme vom Betriebssystemkern getrennt. Damit wird ein grundlegender Selbstschutz des Kerns sowie weiterer laufender Anwendungsprogramme gewährleistet.

Zum Fehlermanagement stellt VxWorks ein entsprechendes Framework bereit, das durch den Nutzer auf seine Bedürfnisse angepasst werden kann. Damit ist eine weitreichende Erkennung und Behandlung von Fehlern in allen Schichten des Betriebssystems möglich. Mit Hilfe der Prozessor-Abstraktionsschicht ist es möglich, VxWorks relativ einfach auf veränderte und neue Hardwarearchitekturen zu portieren. In dieser Schicht werden grundlegende Funktionalitäten gekapselt und von den Eigenschaften der Hardware abstrahiert.

VxWorks ist für folgende Prozessorarchitekturen verfügbar: x86, MIPS, PowerPC, Freescale ColdFire, SH-4, ARM, StrongArm, xScale.

2.4.4 OSEK/VDX und AUTOSAR

Bei AUTOSAR [2] handelt es sich im Gegensatz zu den bisher beschriebenen Betriebssystemen ausschließlich um die Spezifikation von Schnittstellen. Bereits im OSEK[1]/VDX[2]-Standard [43] wurde eine minimale, einheitliche Software-Schnittstelle für Elektronikkomponenten in Automobilen entwickelt. AUTOSAR erweitert OSEK/VDX zu einer vollständigen Middleware-Architektur und Laufzeitumgebung für Automobilanwendungen.

Die OSEK-Initiative wurde 1993 von der deutschen Automobilindustrie ins Leben gerufen. Im Jahr 1994 traten die französischen Automobilhersteller PSA und Renault der Initiative bei und brachten dabei ihren eigenen Standard VDX ein. Das Ziel beider Projekte war die Entwicklung eines Industriestandards für eine offene Architektur für verteilte Steuereinheiten in Kraftfahrzeugen.

Der OSEK/VDX-Standard ist in mehrere Spezifikationen unterteilt. Diese bieten einen Betriebssystemkern und allgemeine Schnittstellen für Kommunikationen, Netzwerkverwaltung und das Debugging. Des weiteren definiert OSEK eine Auszeichnungssprache (*Markup Language*), die *OSEK Implementation Language* (OIL), mit der von Anwendungen benötigte Betriebssystemobjekte beschrieben werden. Weitere Spezifikationen beschäftigen sich mit Erweiterungen für zeitkritische Anwendungen (*Time Triggered Operating System*, TTOS) und fehlertolerante Kommunikation (*Fault Tolerant COMmunication*, FTCOM).

Die *HerstellerInitiative Software* (HIS) [14] hat die Spezifikation für das OSEK-Betriebssystem OSEK OS um einen speziellen *Anwendungsschutz* erweitert. Da-

[1] **O**ffene **S**ysteme und deren Schnittstellen für die **E**lektronik im **K**raftfahrzeug
[2] **V**ehicle **D**istributive e**X**ecutive

mit soll verhindert werden, dass sich Anwendungen verschiedener Hersteller, die auf demselben Knoten laufen, gegenseitig schädigen.

Die verschiedenen OSEK-Spezifikationen sind sehr allgemein gehalten. So ist etwa für die Kommunikation zwischen Anwendungen auch zwischen verschiedenen Steuereinheiten nur ein Protokoll definiert. Keine Beachtung finden hingegen die tatsächlich eingesetzten Kommunikationstechniken wie etwa CAN oder FlexRay.

In AUTOSAR wurden das Ziel eines offenen Standards weiter verfeinert. Wie bereits in OSEK sollten grundlegende Systemfunktionen eine einheitliche Schnittstelle bieten. Durch den zunehmenden Einsatz von elektronischen Steuereinheiten und deren Vernetzung war es nun auch notwendig, die Skalierbarkeit der Software zu gewährleisten. Mit Hilfe der Vernetzung sollte es außerdem möglich sein, Funktionen innerhalb des Fahrzeugnetzwerks zu verschieben. Eng damit verbunden ist auch die Forderung nach der Integration und Austauschbarkeit der Software unabhängig von ihrem Hersteller. So soll vermehrt sogenannte *Commercial-Off-The-Shelf*-Software (COTS Software) zum Einsatz kommen anstelle von für spezielle Anwendungsfälle entwickelte Software. Zuletzt soll durch AUTOSAR die Wartbarkeit über den gesamten Produktlebenszyklus gewährleistet werden, hier insbesondere auch das Einspielen von Softwareaktualisierungen.

Im Jahr 2003 wurde das AUTOSAR-Konsortium nach Vorgesprächen offiziell gegründet. Die erste Spezifikation (Release 1.0) wurde 2005 vorgestellt, 2006 folgte dann Release 2.0. Die „erste serientaugliche Version“ von AUTOSAR wurde im März 2007 als Release 2.1 vorgestellt. Seit August 2008 ist die aktuelle Version mit Release 3.1 verfügbar. In der Zwischenzeit wurde die AUTOSAR-Schnittstelle unter anderem von Elektrobit in deren Toolfamilie *eb tresos* [58] implementiert.

Abbildung 2.8 gibt einen Überblick über den Aufbau einer Steuereinheit.

Die *AUTOSAR Software* besteht aus den einzelnen anwendungsspezifischen Softwarekomponenten, die auf einer ECU ausgeführt werden. Diese kommunizieren über eine einheitliche Schnittstelle (*AUTOSAR Interface*) mit dem *AUTOSAR Runtime Environment* (RTE). Als Abstraktion dieser Kommunikation definiert AUTOSAR den *Virtual Function Bus* (VFB). Diese Kommunikationspfade sind in Abbildung 2.8 als „API 2 VFB & RTE relevant“gekennzeichnet. Bei der Nutzung des *AUTOSAR Interface* kann die Kommunikation zwischen zwei Komponenten auch über ein Netzwerk stattfinden. Die Transparenz dieses Vorgangs wird durch den VFB und das RTE gewährleistet.

Die *Basic Software* stellt die Schnittstelle zu der konkreten ECU dar. Auf dieser Seite werden gegenüber der Laufzeitumgebungen zwei Arten von Schnittstellen bereitgestellt. Zum Einen sind dies wiederum die standardisierten *AUTOSAR Interfaces*, welche auch über das Netzwerk angesprochen werden können. Für bestimmte Komponenten werden dagegen nur einfache *Standardized Interfaces*

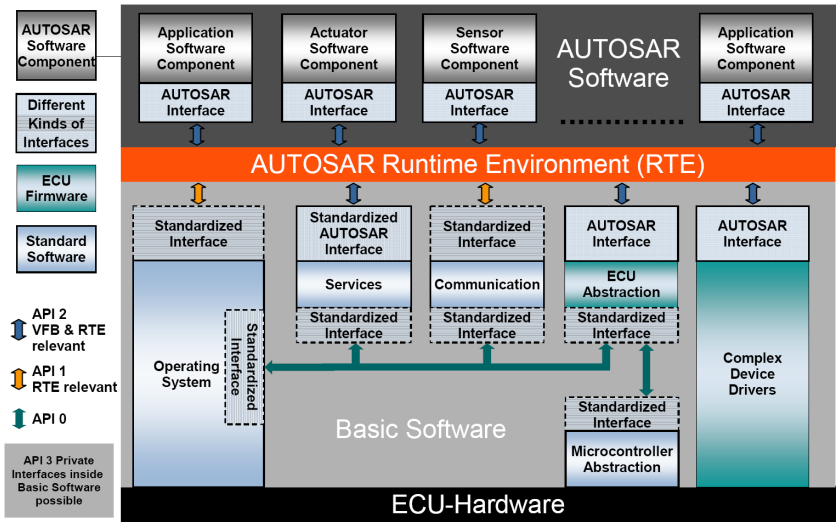


Abbildung 2.8: Überblick über die AUTOSAR-Architektur (aus [3])

definiert, welche meist von der verwendeten Programmiersprache abhängig sind und so nur innerhalb einer ECU angesprochen werden können.

Die *Basic Software* bietet grundlegende Dienste für die AUTOSAR Softwarekomponenten an. Die Komponenten der Basic Software zerfallen in zwei Gruppen. Die ECU Firmware mit *ECU Abstraction* und *Complex Device Drivers* muss speziell für eine bestimmte ECU entwickelt werden, da hier die eigentlichen Hardwarezugriffe abgewickelt werden. Diese Funktionalitäten werden wiederum über standardisierte Schnittstellen der Laufzeitumgebung und der *Standard Software* zur Verfügung gestellt. In der Standard Software werden die Komponenten *Operating System*, *Services*, *Communication* und *Microcontroller Abstraction* zusammengefasst. Diese sind von der darunterliegenden Hardware unabhängig.

Neben der Bereitstellung einer einheitlichen Schnittstelle zur Softwareentwicklung verfolgt AUTOSAR auch die Kapselung von Funktionalitäten. So wird etwa die Kommunikation zwischen einzelnen Steuereinheiten via CAN oder Flexray in separaten Modulen definiert. Dadurch wird zum einen die Fehleranfälligkeit verringert, zum anderen ergibt sich hier auch wieder eine einfachere Analysierbarkeit wie sie für Echtzeitsysteme vonnöten ist.

Die bisherigen Versionen von AUTOSAR waren für den Einsatz auf einfädigen Prozessoren konzipiert. Eine Nutzung mehrfädiger oder gar von Mehrkernprozessoren erweist sich als problematisch [27]. Um das Echtzeitverhalten des höchstprioren AUTOSAR-Tasks zu erhalten, sind einige Erweiterungen an der Spezifikation und ihren Hardwareanforderungen notwendig. Da der Trend im Bereich eingebetteter Systeme im Automobil aber auch zunehmend zum Einsatz

von Mehrkernprozessoren geht, werden zukünftige Versionen von AUTOSAR zumindest eine Unterstützung für diesen Prozessortyp bieten.

Die AUTOSAR-Spezifikation fordert bereits gewisse Schutzmaßnahmen, um die Funktionsfähigkeit eines Systems auch im Fehlerfall zu erhalten. Zum einen ist dies der Einsatz von Speicherschutztechniken, wie sie in heutigen Mikroprozessoren standardmäßig integriert sind. Damit werden die Daten- und Codebereiche von Anwendungen, aber auch in den Adressraum abgebildete Peripheriegeräte geschützt. Auftretende Schutzverletzungen müssen in einem sogenannten `ProtectionHook` verarbeitet werden. Um das Einhalten von Zeitschranken zu gewährleisten, wird ebenfalls eine entsprechende Unterstützung gefordert. So müssen zum einen die Gesamtausführungszeiten von Tasks und Interruptroutinen sowie die Belegungszeiten von Ressourcen bekannt sein. Falls diese überschritten werden, wird wie oben eine Schutzverletzung ausgelöst und durch den `ProtectionHook` behandelt. Zuletzt definiert AUTOSAR auch Mechanismen zum Schutz des Betriebssystems vor schadhaften Anwendungen. Ein grundlegender Dienstschutz war bereits in der OSEK-Spezifikation vorhanden. Dieser wird von AUTOSAR unter anderem durch die Beseitigung undefinierten Verhaltens oder die Einführung „vertrauenswürdiger Anwendungen“ erweitert.

Trumler et al. [59] haben die Erweiterung von AUTOSAR um Selbstkonfigurations- und Selbstheilungstechniken untersucht. Dabei wurde eine Middleware vorgestellt, welche auf der AUTOSAR-Architektur aufsetzt und diese um die entsprechenden Selbstkonfigurations- und Selbstheilungfähigkeiten erweitert. Mit Hilfe der Selbstkonfiguration werden gemäß einer Konfigurationsbeschreibung benötigte Dienste automatisch in einem Netz verteilt, so dass eine optimale Auslastung der einzelnen Knoten erreicht wird. Die Selbstheilung erkennt den Ausfall einzelner Knoten oder Dienste und kann die volle Funktionsfähigkeit des Systems wiederherstellen, indem es die entfallenen Dienste auf anderen Knoten neu startet.

2.4.5 OSA+

Bei OSA+[3] [6, 7] handelt es sich um eine Middleware für verteilte Echtzeitsysteme. Dabei ist ein Einsatz in heterogenen Netzen vorgesehen. Der Entwurf von OSA+ folgt dem aus dem Betriebssystementwurf bekannten Mikrokernel-Paradigma. Der Middlewarekern bringt nur Grundfunktionalitäten mit sich, die zur Verwaltung jeglicher zusätzlichen Programmmodule benötigt werden. Die Gestaltung dieser Grundfunktionalitäten richtet sich an der geringen Leistungsfähigkeit eingebetteter Systeme aus. Ein Einsatz mit erweiterten Funktionalitäten auf leistungsfähigeren Systemen wird dadurch aber nicht ausgeschlossen.

[3] **O**pen **S**ystem **A**rchitecture - **PL**atform for **U**niversal **S**ervices

Zusätzliche Programmmodule stellen ihre Funktionen in Form von Diensten zur Verfügung, weshalb man auch von einer *dienstorientierten Middleware* (*service oriented middleware*) spricht. OSA+ unterscheidet zwei Arten von Diensten: *Basic Services* stellen eine Anpassung der Middleware auf eine bestimmte Hard- und Softwareumgebung dar. Mit optionalen *Extension Services* wird die Funktionalität des Kernsystems erweitert.

Die Kommunikation der Dienste untereinander erfolgt über *Jobs*. OSA+ ist so entworfen, dass es bei Vorhandensein eines Echtzeitbetriebssystems und geeigneter Hardware auch seine eigenen Funktionalitäten in Echtzeitqualität anbieten kann. Aufgrund der Mikrokernelstruktur ist der Kern von OSA+ lediglich 60 kB groß.

Um die zeitliche Vorhersagbarkeit zu gewährleisten, nutzt OSA+ weitgehend Pre-Allokation. Dabei werden bereits beim Laden und Initialisieren eines Dienstes alle von diesem benötigten Ressourcen reserviert. Aufgrund der hier notwendigen Interaktionen hat dieser Vorgang ein unvorhersagbares Zeitverhalten. Sobald aber die Initialisierung abgeschlossen ist, werden nur noch statische Operationen des Dienstes aufgerufen, welche ein konstantes und beschränktes Zeitverhalten bieten.

Wie bereits erwähnt, erfolgt die Kommunikation in OSA+ über sogenannte *Jobs*. Dabei sendet ein Dienst eine Anfrage an einen anderen Dienst und wartet gegebenenfalls auf das Ergebnis. OSA+ unterstützt sowohl synchrone als auch asynchrone Kommunikation. Bei synchronen Jobs ist der anfragende Dienst für die Dauer der Antwortberechnung blockiert. Erfolgt die Anfrage dagegen über die Primitiven zur asynchronen Kommunikation, so kann der anfragende Dienst weiterarbeiten und muss selbstständig eine Antwort abrufen. Für die Abarbeitung der einzelnen Jobs nutzt OSA+ entsprechende Echtzeit-Schedulingverfahren.

Zur weiteren Unterstützung der Echtzeitfähigkeit können außerdem die Dienstgüteinformationen (*Quality of Service, QoS*) der Dienste verwendet werden [46]. Hierzu stellt OSA+ verschiedene Dienstgüteklassen bereit, so etwa auch für die Ausführungszeit eines Dienstes. Der Dienst selbst stellt entsprechende Wertepaare *(Dienstgüteklasse, Wert)* zur Verfügung, aufgrund derer dann eine geeignete Ausführungsreihenfolge für einzelne Jobs festgelegt werden kann.

In manchen Situationen ist eine Rekonfiguration eines Rechenknotens oder Dienstes notwendig. Dies kann etwa der Fall sein, falls durch eine explizite Nutzeranfrage ein neuer Dienst geladen werden soll, oder auch, wenn ein fehlerhafter Dienst ersetzt werden muss. Ebenso ist eine Rekonfiguration bei regelmäßigen Wartungsarbeiten wie etwa dem Einspielen von Softwareaktualisierungen nötig. Dazu wurde ein spezieller Rekonfigurations-Dienst entworfen [53, 8]. Somit ist es möglich, eine Rekonfiguration als normalen Job auszuführen. Somit wird der Middleware-Kern durch die Rekonfiguration nicht blockiert. Aufgrund der Mikrokernstruktur wird der gesamte Code wiederum klein gehalten, da der Rekonfigu-

rationsdienst nur auf jenen Knoten ausgeführt wird, auf denen er auch benötigt wird.

Die Rekonfiguration eines Dienstes zerfällt in drei Phasen. Zunächst wird die neue Version des Dienstes geladen. In der zweiten Phase wird die laufende alte Version durch die neue Version ersetzt. Zuletzt wird die alte Version aus dem System entfernt. Kritisch ist hier die zweite Phase. Hier muss der Zustand der alten auf die neue Version übertragen werden.

- Die einfachste Lösung ist hier, beide Dienste für die komplette Dauer des Transfers zu blockieren. Während der Blockierung werden alle Daten übertragen, danach kann die neue Version des Dienstes ausgeführt werden. Dieses Vorgehen hat eine relativ hohe Ausfallzeit zur Folge.
- Eine Verkürzung der Ausfallzeit wird durch partielle Blockierung erreicht: Hier wird der Zustand des Dienstes kopiert, während die alte Version weiterhin regulär arbeitet. In einer kurzen blockierenden Phase werden schließlich jene Daten erneut übertragen, die sich während des vorherigen Transfers noch geändert haben.
- Zuletzt ermöglicht OSA+ auch eine nichtblockierende Rekonfiguration. Dabei wird eine maximale Ausfallzeit vom Benutzer vorgegeben. In einer nichtblockierenden Phase werden auch hier wieder alle Daten des Dienstes kopiert. Eine blockierende Phase zur Vervollständigung der Daten wird hier dann aber nur angeschlossen, falls diese Übertragung innerhalb der vorgegebenen Ausfallzeit durchgeführt werden kann. Ansonsten wird der komplette Vorgang wiederholt. Der Vorteil einer maximalen Ausfallzeit wird in diesem Fall um den Preis einer im Allgemeinen deutlich längeren Gesamtdauer der Rekonfiguration erkauft.

2.4.6 Vergleich

Tabelle 2.1 gibt einen kurzen Überblick über die in Abschnitt 2.4 vorgestellten Arbeiten und in wie weit diese die hier definierten zusätzlichen Anforderungen bereits erfüllen. Dabei zeigt sich, dass zwar jede Arbeit gewisse Teile der Anforderungen bereitstellt, aber noch keine allumfassende Lösung verfügbar ist.

Aufgrund ihrer Mikrokernstruktur unterstützen bereits mehrere der vorgestellten Betriebssysteme Techniken zur Selbstheilung, so etwa QNX. Hier können anhand der Überwachung kritischer Prozesse Fehler frühzeitig erkannt werden. Durch die modulare Struktur kann das Betriebssystem diese Fehler gut isolieren und die volle Funktionsfähigkeit wiederherstellen.

Den vorgestellten Betriebssystemen fehlen allerdings insbesondere die detaillierte Ermittlung von Laufzeitinformationen (Anforderung RTOS-5/Monitoring) und die feinkörnigen Eingriffsmöglichkeiten nach Anforderung RTOS-6/Reaktionen.

Tabelle 2.1: Vergleich der vorgestellten Echtzeitbetriebssysteme auf die AC-/OC-Anforderungen

Betriebssystem	**RTOS-2'** *Helper Threads*	**RTOS-5** *Monitoring*	**RTOS-6** *System-Aktoren*	**RTOS-7** *Mobiler Code*	**RTOS-8** *Sicherheit*
Choices (2.4.1)	keine Angaben	problemspezifisch	problemspezifisch	nicht verfügbar	Speicherschutz
QNX (2.4.2)	in SMP-Systemen auf eigenem Core möglich	POSIX	POSIX	POSIX Dynamisches Binden	POSIX-Rechteverwaltung, Speicherschutz
VxWorks (2.4.3)	in SMP-Systemen auf eigenem Core möglich	POSIX	POSIX	POSIX Dynamisches Binden	POSIX-Rechteverwaltung, Speicherschutz
AUTOSAR (2.4.4)	als niedrigpriore Tasks ohne jegliche Garantien	-	Hook-Funktionen (bei Bedarf automatischer Aufruf durch OS)	keine explizite Spezifikation, aber möglich	Speicherschutz, Zeitschutz, privilegierte Anwendungen

2.4.7 Zusammenfassung zum Stand der Technik

Die vorgestellten Arbeiten zeigen, dass grundlegende Konzepte des Autonomic und Organic Computing bereits in erheblichem Ausmaß Eingang in die Erforschung und Entwicklung von eingebetteten Systemen gefunden haben. Dabei werden aber zumeist problemspezifische Lösungen entworfen.

Weit verbreitet ist die Nutzung von Speicherschutzmechanismen, um die Code- und Datenbereiche laufender Programme vor gegenseitigem Zugriff zu schützen. Eng damit verbunden ist die Selbstheilungstechnik, das Gesamtsystem bei Absturz eines Programms in einem funktionsfähigen Zustand zu halten (QNX, Abschnitt 2.4.2 sowie VxWorks, Abschnitt 2.4.3). Auch die Selbstheilungsfunktionen, um die Choices erweitert wurde (Abschnitt 2.4.1), greifen vor allem bei fehlerhaften Anwendungen.

Es zeigt sich also, dass bisher das Augenmerk hauptsächlich auf dem Abwenden von negativen Ereignissen liegt. Wenig Beachtung finden dagegen Techniken, die die Leistungsfähigkeit und Nutzbarkeit von Echtzeitsystemen über ein korrektes Verhalten hinaus verbessern. Ebenso findet die zunehmende Vernetzung heutiger eingebetteter Systeme noch wenig Beachtung. Durch die Vernetzung ergeben sich zwar neue Gefahrenquellen für den Betrieb, aber auch neue Möglichkeiten, die Funktionsfähigkeit weiter zu erhöhen. Die Selbst-X-Fähigkeiten stellen dazu entscheidende Wegweiser dar.

2.5 Fazit

Wie bereits in Abschnitt 2.4.7 erläutert wurde, sind die Konzepte des Autonomic und Organic Computing nur teilweise in aktuellen eingebetteten Systemen integriert. Einen ganzheitlichen Ansatz gibt es hierfür allerdings nicht. Dieses Kapitel hat gezeigt, welche Anforderungen für eine vollständige Umsetzung der Konzepte an die Komponenten eines solchen Systems zu stellen sind.

Darauf aufbauend stellt das folgende Kapitel die Architektur eines Echtzeitbetriebssytems vor, welches für die Implementierung eines Organic Managements in eingebetteten Systemen geeignet ist. In den darauffolgenden Kapiteln wird ein Organic Management entworfen, das den besonderen Anforderungen eingebetteter Echtzeitsysteme gerecht wird.

3 Ein Echtzeitbetriebssystem mit Unterstützung für OC

Dieses Kapitel stellt die CAROS-Betriebssystemarchitektur [21] vor. Diese wurde für den Einsatz in Echtzeitsystemen entworfen. Gleichzeitig aber ist sie für Anforderungen aus dem Bereich des Organic Computing vorbereitet. CAROS ist für den Einsatz auf einem eingebetteten mehrfädigen Mikroprozessor konzipiert. Zunächst werden in Abschnitt 3.1 Anforderungen an die Architektur vorgestellt, in Abschnitt 3.2 werden dann die konkreten Implementierungsanforderungen aufgeführt.

3.1 Connective Autonomic Realtime Operating System - CAROS

Der Entwurf der CAROS-Architektur folgt den *Mikrokern*-Prinzipien. Der Betriebssystemkern enthält dabei nur die nötigsten Grundfunktionen wie etwa den Scheduler oder die Ressourcenverwaltung. Jede zusätzliche Funktionalität wird außerhalb des Kerns als eigenständige Komponente ausgeführt, die nur die vordefinierte Schnittstelle zum Kern benutzt. Damit ist es möglich, einzelne Komponenten auszutauschen, ohne dabei die Funktionsfähigkeit anderer Teile des Systems zu beeinflussen. Weiterhin führt ein Fehler in einer Komponente nicht automatisch zum Ausfall des Gesamtsystems. Abbildung 3.1 gibt einen Überblick über den Architekturentwurf für den Betriebssystemkern und essentielle Zusatzkomponenten.

Der Betriebssystemkern besteht aus fünf Komponenten. Grundlage bilden die drei Säulen *Thread Management*, *dynamische Speicherverwaltung* und *Ressourcenmanagement*. Thread Management und Ressourcenmanagement sind auch essentielle Komponenten herkömmlicher Echtzeitbetriebssysteme. Die dynamische Speicherverwaltung erhöht die Flexibilität des Gesamtsystems und kann zur Implementierung komplexerer Selbst-X-Mechanismen verwendet werden. Darauf aufsetzend befindet sich der *Runtime Linker*. Ein *Security Manager* zur Kontrolle der Betriebssystem-Zugriffe vervollständigt die Architektur. Um im normalen Anwendungsbetrieb ein vorhersagbares Zeitverhalten zu garantieren, nutzt CAROS

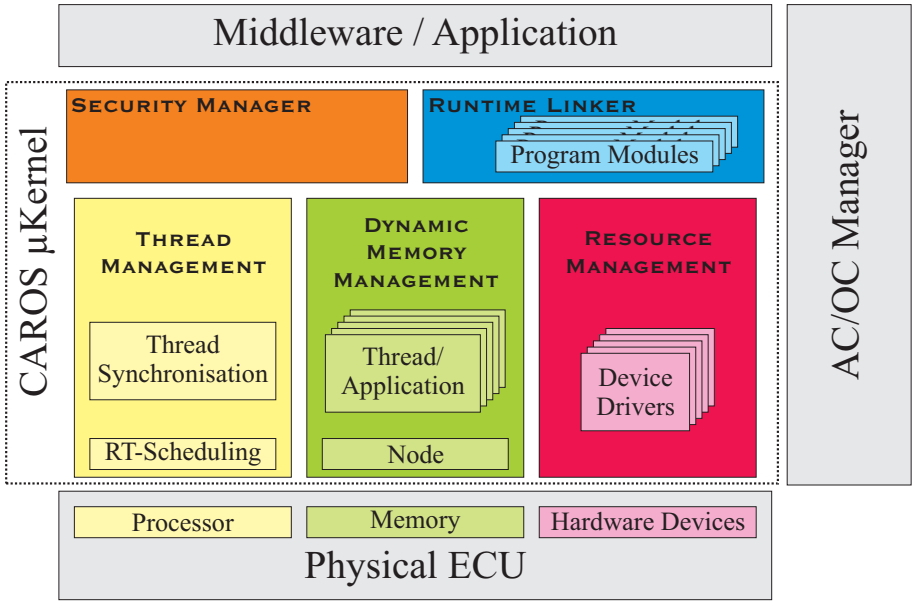


Abbildung 3.1: Architektur des Echtzeitbetriebssystems CAROS

ausgiebig Techniken zur Vorreservierung verschiedener Ressourcen. Der Organic Manager selbst ist nicht Teil des Betriebssystemkerns, allerdings wird seine Implementierung mit Helper Threads durch CAROS unterstützt (siehe unten). In den folgenden Abschnitten werden die fünf Teile des Betriebssystemkerns genauer beschrieben.

3.1.1 Thread Management

3.1.1.1 Scheduler

Der wichtigste Teil der Threadverwaltung ist der Scheduler. Er muss eine echtzeitfähige Schedulingtechnik implementieren. Diese erlaubt es zusätzlich, beliebige Nicht-Echtzeit-Threads parallel zu Echtzeitanwendungen laufen zu lassen. Die Funktionalität und das Zeitverhalten der Echtzeitthreads darf dadurch nicht beeinflusst werden. Dies setzt voraus, dass bereits die unterliegende Hardware eine mehrfädige Programmausführung unterstützt.

Beim Einsatz innerhalb eines verteilten Systems soll CAROS außerdem in der Lage sein, zur Laufzeit neue Anwendungen auf einem Knoten aufzunehmen. Die Zeitschranken dieser Anwendungen sind bei der Integration eines Knotens allerdings noch unbekannt. Deshalb muss es möglich sein, verlässliche Aussagen über die Auslastung und noch freie Rechenkapazität zu treffen. Nur so kann zur

Laufzeit entschieden werden, ob die Aufnahme weiterer Anwendungen überhaupt sinnvoll ist.

Aus diesem Grund empfiehlt sich der Einsatz eines zeitbasierten Schedulingverfahrens. Als Beispiel soll hier das Guaranteed-Percentage-Scheduling [28] dienen. Damit ist es möglich, jedem Echtzeitthread einen konstanten Anteil an der Rechenzeit innerhalb einer sich wiederholenden Zeitperiode zuzuweisen. Die verbleibenden Takte werden zur Ausführung von Nicht-Echtzeitthreads und Helper Threads verwendet. Abbildung 3.2 zeigt schematisch eine solche Periode. Zuerst erhalten die Echtzeitthreads 1-3 ihre Rechenzeitanteile. Nach der Abarbeitung dieser Threads sind noch 40% der Rechenzeit verfügbar. Diese wird nun unter drei Helper Threads gemäß weiterer Vorschriften wie etwa nach gewichtetem *Round Robin* verteilt.

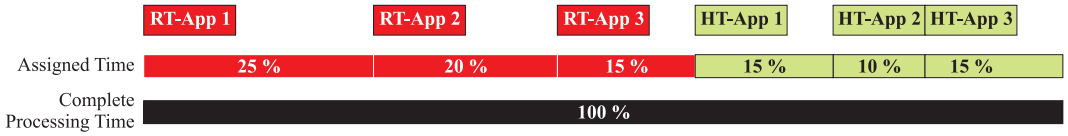


Abbildung 3.2: Die angepasste Guaranteed-Percentage-Schedulingtechnik (RT: Real-Time, HT: non Real-Time Helper Threads)

3.1.1.2 Helper Threads

Helper Threads dürfen das Zeitverhalten eines laufenden harten Echtzeitthreads nicht beeinflussen. Auf einem einfädigen Prozessor laufen sie in den Leerlaufzeiten des harten Echtzeitthreads, müssen aber mit festen Kosten unterbrochen werden können, sobald der harte Echtzeitthread aktiviert wird. Somit laufen die Helper Threads nicht gleichzeitig mit dem harten Echtzeitthread und können so auch nicht die überwachten Threads unterbrechen.

Auf einem mehrfädigen Prozessor können Helper Threads in eigenen Thread Slots parallel zu einem harten Echtzeitthread ausgeführt werden. Dazu wird allerdings ein hardware-basierter Scheduler benötigt, der es erlaubt ohne zusätzliche Kosten zwischen den einzelnen Thread Slots umzuschalten. Nur so ist es möglich, für eine parallel laufende Echtzeitanwendung ein vorhersagbares Zeitverhalten zu garantieren. Bei softwarebasierten Schedulingverfahren müssten hingegen immer gewisse Zeitkosten für jeden Kontextwechsel berücksichtigt werden. Dadurch kann sich die WCET-Abschätzung der harten Echtzeitthreads erhöhen. Helper Threads können auch in gesonderten Kernen eines Mehrkernprozessors laufen.

3.1.1.3 Synchronisation

Mechanismen zur Threadsynchronisation müssen gewöhnlich auch in das Scheduling und die Threadverwaltung eingreifen. Aus diesem Grund sind diese Techniken in das Threadmanagement integriert. Bei der Implementierung ist darauf zu achten, dass die eingesetzten Mechanismen auch für den Echtzeitbetrieb geeignet sind. Zur Vermeidung von Prioritätsinversionen oder Verklemmungen können Techniken wie Prioritätenvererbung und Prioritätsgrenze [56] zum Einsatz kommen. Insbesondere das letztere Verfahren erfordert allerdings eine statische Konfiguration des Gesamtsystems, falls ein bestimmtes Zeitverhalten garantiert werden soll.

3.1.2 Ressourcen-Management

3.1.2.1 Eigenschaften

Eine wichtige Aufgabe jedes Betriebssystems ist die Verwaltung der Hardware-Ressourcen. Gemäß den Mikrokern-Prinzipien verwaltet der CAROS-Kernel nur die wichtigsten Ressourcen direkt. In diesem Fall sind das die Rechenzeit und der Speicher. Alle anderen Ressourcen wie etwa Peripheriegeräte werden von einer dedizierten Ressourcenverwaltung verwaltet. Der Zugriff auf diese Geräte erfolgt durch eigene *Gerätetreiber*, welche den Mikrokern-Prinzipien folgend nicht innerhalb des Kernel, sondern mit den eingeschränkten Zugriffsrechten des User Space ausgeführt werden. Dies betrifft sowohl die generischen `read`/`write`-Aufrufe, also auch treiberspezifische I/O-Operationen. Zugriffe auf ein Gerät (`open`/`close`) sowie dessen Konfiguration (`ioctl`) werden durch den Security Manager kontrolliert, der innerhalb des Kernels läuft.

Das Problem der gleichzeitigen Benutzung einer Ressource durch mehrere Threads kann auf das Problem der Threadsynchronisation mittels Sperrvariablen zurückgeführt werden, wofür von der Threadverwaltung bereits geeignete Maßnahmen bereitgestellt werden.

Gerätetreiber sind nicht statisch an den Kernel gebunden, stattdessen werden sie während des Systemstarts oder zur Laufzeit mit Hilfe des Runtime Linkers geladen. Damit ist es möglich, bei Bedarf Gerätetreiber zur Laufzeit auszutauschen und zu aktualisieren.

Diesen Kriterien folgend stellt die Ressourcenverwaltung eine wichtige Grundlage für Selbstkonfigurationstechniken dar. Die Fähigkeit, einen Treiber zur Laufzeit zu ersetzen, ermöglicht eine hohe Adaptivität des Systems.

Die Gerätetreiber selbst müssen dem Betriebssystem zumindest rudimentäre Zustandsinformationen über den funktionellen Zustand bereitstellen. Um ein Organic Management besser zu unterstützen, können auch aufwändigere Monitoring-Funktionen implementiert werden, welche dann genauere Informationen liefern.

3.1.2.2 Überlegungen zum Echtzeitverhalten

Im Allgemeinen ist die Anzahl der Treiber unbegrenzt, die durch die Ressourcenverwaltung unterstützt werden. Dies führt allerdings zwangsläufig zum Einsatz dynamischer Datenstrukturen innerhalb der Verwaltung. Beim Zugriff auf solche Strukturen kann unter Umständen kein beschränktes Zeitverhalten garantiert werden. Für die Nutzung in Echtzeit-Anwendungen muss die Ressourcenverwaltung den Zugriff auf die Treiber in konstanter Zeit ermöglichen. Art und Anzahl der Geräte, die eine Anwendung nutzt, sind begrenzt und im Voraus bekannt. Die Geräte-Handler können deshalb bereits bei der Vorbereitung der Ausführungsumgebung der Anwendung bereitgestellt werden. Im Falle statisch gebundener Anwendungen erfolgt dies bereits beim Systemstart. Die Handler von zur Laufzeit geladenen Anwendungen werden direkt im Anschluss an den Einbindungsprozess bereitgestellt. Damit können Gerätezugriffe in konstanter Zeit durchgeführt werden. Der Zugriff für Nicht-Echtzeitanwendungen kann hingegen über eine dynamische Namensauflösung erfolgen.

3.1.3 Dynamic Memory Management

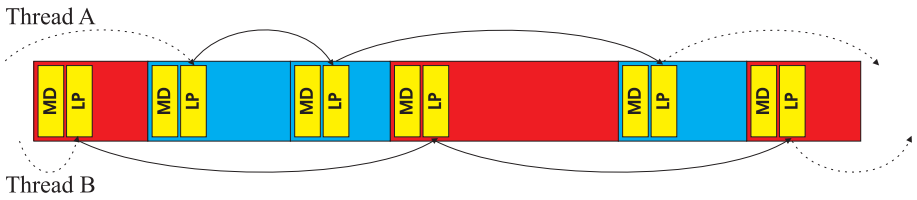
Oberstes Ziel der Speicherverwaltung ist die Trennung der laufenden Threads auf Speicherebene. Außerdem soll sie auch Echtzeitanwendungen neue Möglichkeiten bieten, weshalb auch die Speicherverwaltung echtzeitfähig sein muss. CAROS erreicht dies durch die Einführung einer zweischichtigen Speicherverwaltung mit *a priori*-Allokation.

Die erste Ebene, die *Knotenebene*, weist den einzelnen Threads große Speicherblöcke zu. Diese Zuweisungen müssen durch Sperren gesichert werden, es können hier also Blockierungen auftreten. Für harte Echtzeitthreads wird diese Zuweisung deshalb nur einmalig bei deren Initialisierung durchgeführt und hat somit keinen Einfluss auf deren Verhalten während des Echtzeitbetriebs. Die Folgen möglicher Blockierungen werden außerdem durch die echtzeitfähigen Synchronisationstechniken des Thread Managements abgeschwächt.

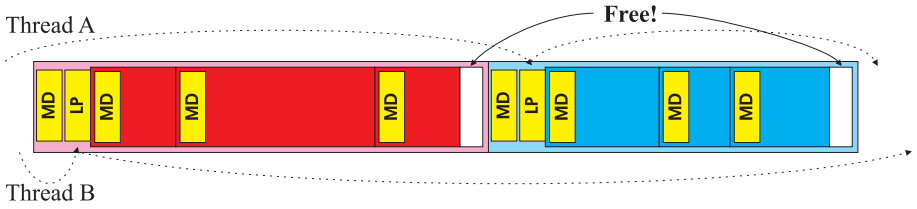
Die zweite Ebene, die *Threadebene*, weist nun dem innerhalb des Threads laufenden Programm auf dessen Anforderung hin Speicher zu. Diese Zuweisung ist *blockierungsfrei*, da der Speicher aus den vorher bereitgestellten großen Blöcken

entnommen wird, auf welche *per definitionem* nur der zugehörige Thread zugreifen darf.

Ein weiterer Vorteil ist aus Abbildung 3.3 ersichtlich. Das Betriebssystem muss beim Beenden eines Threads sicherstellen, dass es dessen gesamten reservierten Speicher wieder freigibt. Dazu hält es die einzelnen Speicherblöcke eines Threads in einer Liste (Listenzeiger LP). Wie Abbildung 3.3(a) zeigt, muss bei der herkömmlichen Speicherverwaltung bei jedem kleinen Block ein solcher Listenzeiger angelegt werden. Im Falle der zweistufigen Verwaltung hingegen müssen diese Zeiger nur zu den großen Blöcken hinzugefügt werden, die auf Knotenebene bereitgestellt werden, wie es in Abbildung 3.3(b) gezeigt wird. Bereits in diesem einfachen Beispiel führt dies zu einer Reduzierung des Speicherverbrauchs. Damit wird auch der höhere Aufwand aufgewogen, der durch das Führen mehrerer Verwaltungsdatenstrukturen entsteht. Das Aufräumen bei Beendigung eines Threads wird erheblich erleichtert, da nur wenige große Speicherblöcke freigegeben werden müssen.



(a) Herkömmliche Speicherverwaltung: die Speicherblöcke der Threads A und B sind vermischt



(b) Zweistufige Speicherverwaltung: die äußeren Kästen stellen die Blöcke der Knotenverwaltung dar

Abbildung 3.3: Beispielhafte Speicherbelegung mit zwei Threads; MD: Managementdaten der Speicherverwaltung, LP: Zeiger zur Verkettung der Speicherliste eines Threads

3.1.4 Runtime Linker

Der *Runtime Linker* ist eine vielseitig eingesetzte Komponente. Zum einen ist er die Voraussetzung, um zur Laufzeit neue Programmmodule zu laden. Er wird aber auch von der Ressourcenverwaltung benötigt, um Gerätetreiber zu laden.

Der kompilierte Code solcher Module enthält im Allgemeinen noch symbolische Referenzen zu Funktionen des Betriebssystems oder anderer Module. Bei absolut adressierten Sprüngen und Datenzugriffen müssen außerdem noch die Zieladressen eingetragen werden, da diese von der genauen Lage des Moduls im Speicher abhängen. All diese Arbeiten werden von dem Runtime Linker übernommen, wenn ein Modul auf einen bestimmten Knoten geladen wird.

Aufgrund der symbolischen Referenzen ist der Bindungsprozess selbst nicht echtzeitfähig, sondern hängt stark von deren Art und Anzahl sowie den zur Auflösung benutzten Datenstrukturen ab. Allerdings kann der Bindungsprozess im Hintergrund als Helper Thread ablaufen und so zumindest indirekt den Echtzeitbetrieb des Systems unterstützen [44].

Aus Sicherheitsgründen muss das Betriebssystem auch einen Zugriffsschutz für dynamisch gebundene Module bereitstellen. Auf Symbole (Funktionen und Variablen), die von Programmmodulen bereitgestellt werden, darf nicht global durch alle Anwendungen auf einem Knoten zugegriffen werden können. Stattdessen soll der Zugriff auf die Symbole eines Moduls zunächst auf jene Anwendung beschränkt sein, die das Modul auch geladen hat. Diese Anwendung kann wiederum weiteren, ausgewählten Anwendungen Zugriff auf ihre Symbole gewähren. CAROS stellt dazu *Namensräume* bereit, über welchen es Zugriffe auf Symbole auflöst. Ein globaler Namensraum stellt alle Symbole der Betriebssystemschnittstelle zur Verfügung, kann aber auch zusätzliche grundlegende Bibliothekssymbole enthalten. Jede Anwendung besitzt zudem einen privaten Namensraum, auf den nur sie zugreifen kann. Auf diese Weise werden diese privaten Symbole vor Manipulationen durch andere Anwendungen geschützt.

Zur effizienten Nutzung des Arbeitsspeichers sollte es auch eine Möglichkeit geben, Programmmodule aus dem laufenden System zu entfernen. Dies betrifft insbesondere den Fall, in dem ein Programmmodul durch eine neuere Version ersetzt wird, oder eine Anwendung auf einen anderen Knoten migriert wird. Der Speicher, den die alte Version beziehungsweise die verschobene Anwendung belegt, muss wieder freigegeben werden. Der Runtime Linker muss in all diesen Fällen die Konsistenz der geladenen Module sicherstellen. Insbesondere dürfen keine Module entfernt werden, auf die von anderen geladenen Modulen oder Programmen noch zugegriffen werden könnte.

3.1.5 Security Manager

Die vorgestellten Dynamisierungstechniken erlauben einen flexiblen Einsatz von Steuereinheiten. Sie bergen aber auch das Risiko, dass das Systemverhalten zur Laufzeit unvorhersehbar beeinflusst wird. So ist es prinzipiell möglich, auf einem Knoten unkontrolliert neue Anwendungen zu starten, was zu einer Überlastung

des Prozessors und im schlimmsten Fall zum Ausfall wichtiger Anwendungen führen kann. Ebenso kann eine zu ausgiebige oder unkontrollierte Nutzung des Arbeitsspeichers andere Anwendungen bis zur Funktionsunfähigkeit beeinträchtigen.

Um solche und ähnliche Situationen zu vermeiden, stellt CAROS eine Rechteverwaltung bereit. Beim Aufruf eines Betriebssystemdienstes wird zunächst geprüft, ob die aufrufende Anwendung überhaupt über die entsprechenden Rechte für den Dienst verfügt. Außerdem verfügt der CAROS Security Manager über ein kohärentes System zur Weitergabe von Rechten, etwa wenn eine Anwendung eine andere startet.

Ein weiterer Aspekt ist die sichere Kommunikation mit anderen Knoten. Der Kern selbst enthält keine Kommunikationskomponente und kann somit nicht direkt eine sichere Kommunikation mit anderen Knoten unterstützen. Stattdessen stellt der Sicherheitsmanager mit Hilfe geeigneter Module sichere Kommunikationskanäle zur Verfügung. Er verschlüsselt versendete Daten, um deren Manipulation oder Abhören zu verhindern. Außerdem kann er Kommunikationsprotokolle implementieren, die einen Datenaustausch unter Echtzeitbedingungen erlauben.

3.2 Implementierung von CAROS

Die beschriebene CAROS-Architektur wurde auf dem CarCore-Prozessor (Abschnitt 2.2.3) implementiert. Die folgenden Abschnitte beschreiben diese Implementierung und zeigen wichtige Entscheidungen auf, die während dieser Implementierung getroffen wurden.

3.2.1 Thread Management

Die Tatsache, dass die Zielplattform bereits einen vollwertigen Scheduler bereitstellt, vereinfacht die Implementierung des Thread Managements erheblich. Im Wesentlichen stellt CAROS damit eine komfortable Schnittstelle zu den Registern und TCBs des Hardware-Schedulers bereit. CAROS unterstützt dabei das Anlegen und Beenden von Threads zur Laufzeit, eine statische Konfiguration findet in diesem Bereich nicht statt. Um trotzdem die Echtzeitfähigkeit zu gewährleisten, muss die maximale Anzahl der DTS- und PIQ-Threads bei der Kompilierung festgelegt werden. Für diese Threads werden alle Systemressourcen wie etwa Stack, Threadslots etc. statisch reserviert, nur die Zuweisung erfolgt zur Laufzeit. Damit kann CAROS auch für das Anlegen neuer Echtzeit-Threads eine beschränkte Ausführungszeit garantieren.

Synchronisation Zur Thread-Synchronisation erlaubt CAROS die Nutzung von Sperr- und Bedingungsvariablen (*Mutex*, *Conditional*), wie sie auch im POSIX-Standard [47] vorgesehen sind. Weitere Synchronisationsprimitive wie etwa *Barrieren* lassen sich aus den Vorhandenen ableiten.

Auf die Integration von Prioritätenvererbung, wie sie etwa in OSEK und AUTOSAR zum Einsatz kommt, verzichtet die Implementierung. Aufgrund der zeitbasierten Schedulingstrategie kann innerhalb der Threadklassen DTS (harte Echtzeit) und PIQ (weiche Echtzeit) keine Prioritäteninversion auftreten. Eine Prioritäteninversion entsteht normalerweise durch das Zusammenspiel von drei Threads T_H, T_M, T_L mit hoher, mittlerer und niedriger Priorität, wie Abbildung 3.4 zeigt. Die Threads T_H, T_L mit hoher und niedriger Priorität greifen dabei auf eine gemeinsam genutzte Ressource zu. Der mittelpriore Thread T_M hat keinerlei Interaktion mit den beiden anderen Threads. Falls nun der niedrigpriore Thread T_L die gemeinsam genutzte Ressource hält, wird T_H unterbrochen, sobald er auf dieselbe Ressource zugreifen muss. Dieser Sachverhalt ist aber bereits bei der Systemintegration bekannt und kann berücksichtigt werden. Durch Ereignisse zur Laufzeit kann es nun aber passieren, dass der mittelpriore Thread T_M aktiviert wird, während T_L die Ressource hält und T_H auf die Freigabe dieser wartet. Da T_M eine höhere Priorität als T_L besitzt, wird er vor diesem ausgeführt. Damit wird aber die Freigabe der Ressource verzögert und letztendlich T_H von T_M blockiert, was einer Umkehrung ihrer Prioritäten entspricht.

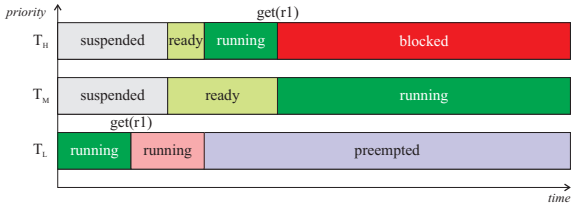


Abbildung 3.4: Zeitlicher Ablauf einer Prioritäteninversion

Im CarCore hingegen stellt sich diese Problematik nicht. Aufgrund des zeitbasierten Schedulingverfahrens (DTS und PIQ) wird auch für T_L immer ein Fortschritt garantiert, T_L kann also nicht durch T_M beeinflusst werden. Bei der Systemintegration muss daher nur die maximale Belegungsdauer der Ressource bestimmt werden, es können aber keine unvorhergesehenen Einflüsse durch andere Threads auftreten.

Beeinflussungen durch Nicht-Echtzeitthreads (RRIQ) können dadurch allerdings nicht ausgeschlossen werden. Deshalb muss bei der Entwicklung eines Systems auf eine gemeinsame Nutzung derselben Ressourcen durch Echtzeit- und Nicht-Echtzeitthreads verzichtet werden. Ebenso verhindert das zeitbasierte Scheduling keine Verklemmungen, die durch verschachtelte Ressourcenbelegungen entstehen können. Bei einer statischen Konfiguration des Systems können Verklemmun-

gen durch einen geeigneten Software-Scheduler verhindert werden. Ein mit dem Scheduler verknüpfter *Task Filter* [27] gibt nur solche Tasks zur Ausführung frei, die sich keine Ressourcen teilen. Dabei kann allerdings maximal ein Thread unter Echtzeitbedingungen laufen. Falls dagegen wie im CarCore die gleichzeitige Ausführung von mehreren Echtzeitthreads möglich ist, muss auf verschachtelte Ressourcenbelegungen verzichtet werden. Eine entsprechende Einschränkung findet sich auch in Arbeiten zur Ressourcenverwaltung in Multiprozessorsystemen [49, 4].

3.2.2 Dynamische Speicherverwaltung

Zur dynamischen Speicherverwaltung werden zwei Verfahren verwendet. Zum einen ist dies der *Best-Fit-Allocator* von Doug Lea [30]. Dieser erlaubt eine effiziente Nutzung des verwalteten Speichers mit geringer interner Fragmentierung der bereitgestellten Blöcke, d.h. Speicheranforderungen werden mit Blöcken möglichst passender Größe beantwortet. Dieses Verfahren ist allerdings nur sehr eingeschränkt echtzeitfähig, da die Laufzeit jeder Speicheranforderung und -freigabe sehr stark von den vorherigen Anforderungen und Freigaben abhängt.

Deshalb kommt als weiteres Verfahren *Two-Level-Segregate-Fit* (TLSF) zum Einsatz. Dieses Verfahren bietet eine beschränkte Ausführungszeit und ist damit echtzeitfähig [35]. Die Echtzeitfähigkeit wird um den Preis einer höheren internen Fragmentierung erkauft, d.h. die bereitgestellten Blöcke sind im Allgemeinen größer als die Anforderung.

Beiden Verfahren ist gemein, dass der Zuweisungs- und Freigabeprozess auf einem Speicherbereich exklusiv ausgeführt werden muss. Es kann also nur eine Instanz einer Speicherzuweisung oder -freigabe ausgeführt werden, nie aber beides gleichzeitig oder in mehreren Instanzen. Problemlos ist hingegen die parallele Ausführung mehrerer Instanzen, wenn diese getrennte Speicherbereiche verwalten.

Für die Echtzeitfähigkeit des Systems ergeben sich damit diese Folgen: Wie bereits unter 3.2.1 erläutert, werden die Systemressourcen für Echtzeitthreads statisch bereitgestellt. Dies gilt allerdings nicht für den Haldenspeicher (*Heap*) eines Threads. Sollte es notwendig sein, einen Echtzeitthread mit der Fähigkeit zur dynamischen Speichernutzung auszustatten, so muss das Zeitverhalten der Knotenebene der Speicherverwaltung in die Zeit zum Anlegen des Threads einbezogen werden. Das Anlegen eines neuen Threads ist damit grundsätzlich nicht echtzeitfähig. Zur Laufzeit hingegen garantiert CAROS die Echtzeitfähigkeit, indem es Echtzeitthreads nur die Nutzung des echtzeitfähigen TLSF-Allokators erlaubt. Problematisch ist es allerdings, wenn der Haldenspeicher eines Echtzeitthreads zur Laufzeit vergrößert werden muss. Hier wird auf die Knotenebene der Spei-

cherverwaltung zugegriffen, wodurch es zu nicht vorhersagbaren Blockierungen kommen kann. Deshalb wird für harte Echtzeitthreads bei deren Initialisierung ein nach Maßgabe des Entwicklers ausreichend großer Block vorreserviert, eine Erweiterung zur Laufzeit ist nicht möglich.

3.2.3 Ressourcenverwaltung

Die Schnittstelle der Ressourcenverwaltung orientiert sich stark an den Vorgaben des POSIX-Standards [47]. CAROS stellt Funktionen zur Reservierung und Freigabe eines Geräts (`open`, `close`) sowie dessen Konfiguration (`ioctl`) bereit. Die generischen I/O-Operationen `read, write` werden nach der Zugriffskontrolle durch den Betriebssystemkern direkt an den Gerätetreiber weitergegeben.

Die Gerätetreiber selbst werden mit Hilfe des Runtime-Linkers geladen. Damit ist es auch möglich, einen Treiber während des Betriebs auszutauschen, ohne das komplette System anhalten zu müssen.

Weiterhin bietet die Ressourcenverwaltung die Möglichkeit, Gerätetreiber mit einer geeigneten Selbstbeschreibung auszustatten. Falls ein angefordertes Gerät nicht verfügbar ist, kann der Anwendung mit Hilfe der Selbstbeschreibung ein Ersatzgerät zugewiesen werden. Dadurch wird die Funktionsfähigkeit der Anwendung zwar eingeschränkt, aber ein Totalausfall verhindert.

3.2.4 Sicherheit

Die Zugriffsschutzfunktionalität des *Security Manager* läuft über den ganzen Betriebssystemkern verteilt, es gibt kein dediziertes Sicherheitsmodul. Jeder Zugriff auf den Kern wird zunächst einer Rechteprüfung unterworfen, und nur wenn diese erfolgreich ist, führt der Kern den Zugriff auch aus.

Zugriffsrechte können jeweils für Gruppen verwandter Funktionen gesetzt werden. So existieren etwa Gruppen für die Threadverwaltung (Anlegen und Bearbeiten von Threads), oder die dynamische Speicherverwaltung. Eine Weitergabe von Zugriffsrechten an andere Threads ist nur auf Basis des Vererbungsprinzips möglich, d.h. ein Thread kann nur solche Rechte weitergeben, die er auch selbst besitzt.

3.2.5 Runtime-Linker

Grundlage für mobilen Code und Gerätetreiber sind die Objektdateien (`.o`), die mit den TriCore-Compilern generiert werden können. Weder der TriCore-GCC noch der TriCore-Compiler von Tasking unterstützen die Erstellung dynamischer

Bibliotheken (`.so`). Um ein geeignetes Format der Objektdateien sicherzustellen, bietet CAROS spezielle Header-Dateien für den Entwurf von Applikationen oder Gerätetreibern an.

Der Runtime-Linker erstellt aus diesen Objektdateien ein lauffähiges Speicherabbild des Codes. Dieses wird dann von weiteren Verwaltungskomponenten wie etwa der Ressourcenverwaltung in den Betrieb eingebunden.

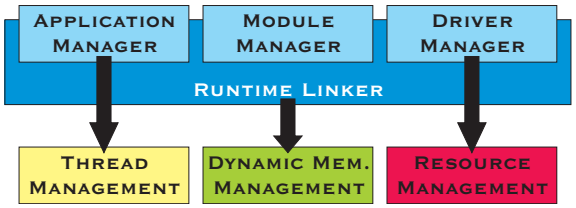


Abbildung 3.5: Runtime-Linker mit seinen Modulen und deren Interaktion mit dem Betriebssystemkern

Der Runtime-Linker selbst kann von Anwendungsseite aus nicht direkt genutzt werden. Wie Abbildung 3.5 zeigt, stellt er drei Komponenten *Application Manager*, *Module Manager* und *Driver Manager* als Schnittstellen zur Verfügung. Der Linker selbst nutzt stark die dynamische Speicherverwaltung, da der endgültige Speicherbedarf eines Programmmoduls erst während des Bindungsprozesses feststeht. Eine statische Speicherallokation ergäbe hier keinen Sinn. Der *Application Manager* bietet ein spezielles Framework, das es erlaubt Anwendungen zur Laufzeit anzuhalten, auf andere Knoten zu verlagern und dort deren Ausführung fortzusetzen [22]. Damit erhöht er die Adaptionsfähigkeit eines verteilten Systems von CAROS-Knoten. Er arbeitet dabei eng mit der Threadverwaltung zusammen, die in Form von Threads Container für die Anwendungen bereitstellt. Der *Module Manager* verwaltet Programmmodule, die einer oder mehreren Anwendungen Funktionalitäten zur Verfügung stellen, aber nicht als eigenständige Anwendung ausgeführt werden. Der *Driver Manager* lädt Gerätetreiber und bindet diese in die Ressourcenverwaltung ein.

3.3 Fazit

Die CAROS-Architektur sowie deren Implementierung in der hier vorgestellten Form bilden die Grundlage für die Integration von Autonomic/Organic-Computing-Techniken in eingebetteten Echtzeitsystemen. Den Anforderungen an ein Echtzeitbetriebssystem (Abschnitt 2.1.1) trägt CAROS vor allem durch sein Thread Management Rechnung. Es erlaubt eine mehrfädige Programmausführung unter Verwendung eines echtzeitfähigen Schedulingverfahrens. Dazu

nutzt es das DTS-Scheduling des CarCore-Prozessors. Ebenso stellt CAROS vorhersagbare Synchronisationsmechanismen zur Verfügung. Der CAROS-Mikrokern ist so entworfen, dass sein Zeitverhalten vorab bestimmt werden kann. Wo dies nicht möglich ist, kann die entsprechende Funktionalität abgetrennt von der Echtzeitanwendung in Form eines Helper Threads ausgeführt werden, was auch eine Anforderungen aus Sicht des Autonomic/Organic Managements ist. Auch die weiteren Anforderungen bezüglich der Integration von AC-/OC-Techniken erfüllt CAROS. Die Kernel-Komponenten sind so entworfen, dass sie ein Autonomic Management mit aktuellen Statusinformationen versorgen können, und sie erlauben auch entsprechende Eingriffe in ihr Verhalten. Der Runtime Linker bietet die notwendige Unterstützung für Codemigration, so dass ein Autonomic Management innerhalb eines verteilten Systems Anwendungen zwischen Knoten verschieben kann. Zuletzt verfügt CAROS auch über Sicherheitsmaßnahmen, die die Funktionsfähigkeit des Gesamtsystems sicherstellen können. Diese sind das zeitbasierte Schedulingverfahren und die zweischichtige dynamische Speicherverwaltung zur Isolation der Anwendungen, sowie der Security Manager, der insbesondere die Kernelzugriffe der Anwendungen kontrolliert. Damit bildet CAROS die Grundlage für die Implementierung eines Autonomic Managements.

4 Zweischichtige Management-Architektur

Dieses Kapitel behandelt den Entwurf einer grundlegenden Architektur für ein *Autonomic Management* in eingebetteten Echtzeitsystemen [24]. Diese Architektur baut auf den Grundarchitekturen für Autonomic Computing und Organic Computing (Kapitel 2) auf. Basierend auf verschiedenen Entwurfsmustern wird ein Kommunikationsmodell für das *Autonomic Management* entworfen.

Im Rahmen der Anwendungsbereiche dieser Arbeit, also insbesondere im Bereich der Automobil-Elektronik, müssen verschiedene Randbedingungen beachtet werden, die durch Einschränkungen der Hardware sowie aus besonderen Anforderungen der Anwendungen entstehen. Insbesondere gilt für harte Echtzeitanwendungen, dass diese in keinem Fall ihre Zeitschranken verpassen dürfen. Jegliche AC-/OC-Management-Funktionalität darf also das Zeitverhalten der Hauptanwendung nicht in unvorhersagbarer Weise beeinflussen.

Die Microcontroller und die Software sind im Allgemeinen möglichst gut aufeinander abgestimmt, um eine möglichst hohe Auslastung des Prozessors zu erreichen und die Kosten niedrig zu halten. Da aber in harten Echtzeitanwendungen die Worst Case Execution Time maßgeblich für die Dimensionierung eines Prozessors ist, und die WCET im allgemeinen nicht voll ausgenutzt wird, kann man davon ausgehen, dass ein kleiner Anteil der Rechenzeit für andere Aufgaben zur Verfügung steht. Dies kann dann auch ein AC-/OC-Management sein, welches die Echtzeitarbeit eines solchen Systems unterstützt. Dazu muss es schnelle Reaktionen mit festem Zeitaufwand bereitstellen. Dieser Zeitaufwand muss außerdem eine niedrige obere Schranke (im Sinne von ausgeführten CPU-Takten) besitzen.

Die Reflexe von Lebewesen erfüllen ähnliche Aufgaben. Sie erlauben eine schnelle Reaktion, gewöhnlich um in bestimmten Situationen Schaden von dem Lebewesen abzuwenden. Als Beispiel sei hier der Reflex des Augenlids genannt, durch den der Augapfel vor Schmutz und Austrocknung geschützt wird. Verschiedene Reflexe unterscheiden sich in der Art, wie der auslösende Reiz in eine entsprechende Reaktion umgesetzt wird. Aber allen ist gemeinsam, dass diese Umsetzung automatisch und ohne bewusstes Nachdenken erfolgt.

Erst im Anschluss an die Reflexhandlung, wenn überhaupt, übernimmt der bewusste Teil des Gehirns und führt eine genauere Analyse der Situation durch. Dabei werden dann umfangreichere Umgebungsinformationen ausgewertet. Auf diesem Weg wird eine leistungsfähigere und weitsichtigere Reaktion abgeleitet, die möglicherweise auch die zuvor ausgeführte Reaktion revidiert.

4.1 Grundarchitektur

Auch in einem eingebetteten System steht der Systemüberwachung eine große Anzahl von Sensordaten zur Verfügung. Gleichzeitig gibt es viele Punkte, an denen mit entsprechenden Reaktionen in das Systemverhalten eingegriffen werden kann. Die vollständige Auswertung aller Sensordaten und Parameter gestaltet sich allerdings als zu aufwändig. Oft ist es schon möglich, basierend auf einem relativ kleinen Parametersatz eine geeignete Lösung für ein bestehendes Problem zu finden, wie die Parallele der Reflexhandlung in der Natur zeigt. Bessere Lösungen mögen bei der Betrachtung einer größeren Parametermenge zwar durchaus denkbar sein. Durch den Einsatz komplexer Planungs- oder Lernalgorithmen würde sich allerdings die Reaktionszeit beträchtlich und möglicherweise ohne obere Schranke erhöhen.

Das folgende Beispiel soll diese Idee verdeutlichen. In der Ausgangssituation überwacht ein Watchdog-Dienst eine weiche Echtzeitanwendungen auf ihren Fortschritt. Falls er nun wiederholte Überschreitungen der Deadline feststellt, sind verschiedene Gegenmaßnahmen denkbar. Die einfachste Lösung ist, den Rechenzeitanteil des Anwendungsthreads zu erhöhen. Dazu müssten nur zwei Parameter überwacht werden: das Zeitverhalten der Anwendung (Verpassen der Deadlines), sowie die aktuell von allen Echtzeitanwendungen belegte Rechenzeit des Prozessors. Weiterhin ist nur ein Parameter von der Reaktion betroffen, eben der Rechenzeitanteil des Anwendungsthreads. Eine weitere Lösung ist es, die Taktfrequenz des Prozessors zu erhöhen. Damit würde allerdings auch die Leistungsaufnahme des Prozessors beeinflusst sowie möglicherweise das Zeitverhalten anderer Anwendungen, was den zu betrachtenden Parametersatz schon vergrößern würde. Zuletzt können auch die Systemüberwachungsdaten mehrerer ECUs zusammengeführt werden. Auf Grundlage dieser Daten ist es möglich, über eine Verlagerung der Anwendung auf eine leistungsstärkere ECU mit mehr freier Rechenzeit zu entscheiden. Insbesondere für die letzte Lösung muss aber eine deutlich größere Datenbasis ausgewertet werden.

Aus diesem Grund sollen Selbstverwaltungsentscheidungen auf zwei Ebenen getroffen werden (siehe Abbildung 4.1). Auf der unteren Ebene sind kompakte Verwaltungseinheiten (*Modulmanager*) direkt an einzelne Hardware- oder Softwarekomponenten angebunden. Diese Modulmanager arbeiten auf einem kleinen

Parametersatz und enthalten nur einen kleinen und beschränkten Regelsatz. Sie besitzen Aktoren, die wiederum direkt auf den angebundenen Hardware- oder Softwarekomponenten arbeiten. Durch diesen Aufbau können die Modulmanager ihre Entscheidungen schnell sowie innerhalb eines nach oben beschränkten Zeitraums treffen. Diese Modulmanager entsprechen den Reflexen in der Natur, welche ebenfalls nur einen oder wenige Reize zu ihrer Auslösung benötigen, und dafür eine schnelle Reaktion liefern. Solche einfachen Modulmanager lassen sich so entwerfen, dass sie ihren Reflex in beschränkter Zeit ausführen. Je nach Art und Anwendung können solche Modulmanager direkt in Echtzeitanwendungen integriert werden. Damit ist die gesamte Schicht des Modulmanagements echtzeitfähig.

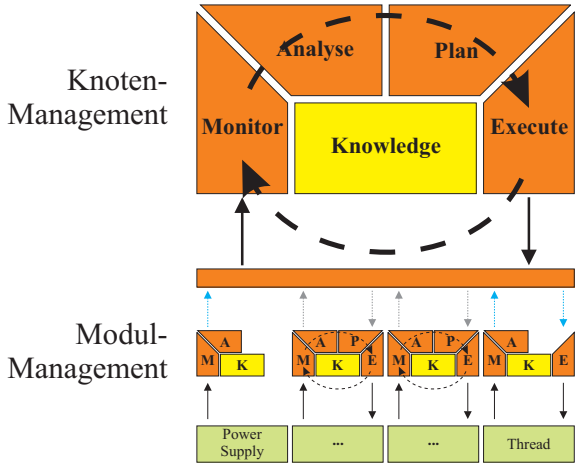


Abbildung 4.1: Zweischichtige OC-Management Architektur; die lokale Managementkomponente ganz rechts implementiert keinen eigenen Planer, sondern muss vom Globalen Manager verwaltet werden

Nachdem ein Modulmanager seine Reaktion ausgeführt hat oder falls er keine geeignete Reaktion für einen bestimmten Zustand finden kann, führt er eine Vorauswertung der überwachten Parameter durch und gibt diese Daten weiter an den *Knotenmanager*. Durch die Vorauswertung werden alle Daten aus der Systemüberwachung in eine gemeinsame Domäne abgebildet. Dadurch können der Knotenmanager und die ihm innewohnenden Algorithmen möglichst generisch entworfen werden. Des weiteren stellen die Modulmanager dem Knotenmanager ihre Aktoren bereit, mit denen dieser Reaktionen ausführen kann.

Der Knotenmanager führt die Überwachungsinformationen aller Modulmanager zusammen, wie dies auch im Gehirn von Lebewesen geschieht. Er übernimmt die Kontrolle, wenn eine globale Sicht auf alle Module notwendig ist, um eine Entscheidung zu treffen. Dazu sind nun mehrere Techniken denkbar. Zum einen könnten ein Klassifizierungssystem oder generische Regeln eingesetzt wer-

den. Aber auch mit Hilfe eines Planers können komplexe Reaktionsfolgen hergeleitet werden. Durch den Einsatz einer dieser Techniken leitet der Knotenmanager die notwendigen Reaktionen für den aktuellen Systemzustand her und führt diese dann mit Hilfe der von den Modulmanagern bereitgestellten Aktoren aus.

Es ist auch möglich, dass ein Modulmanager keine eigene Entscheidungsinstanz implementiert. In Abbildung 4.1 ist dies der Modulmanager am rechten Rand. In einem solchen Fall wird eine Entscheidung immer vom Knotenmanager getroffen. Ebenso können Modulmanager als Monitore mit Vorauswertung (MA- -) oder reine Aktoren (- - -E) implementiert werden. Reine Monitore (M- - -) sind dagegen nicht möglich, da immer eine Vorauswertung hinsichtlich der generischen Schnittstelle des Knotenmangers notwendig ist (linker Modulmanager).

4.2 Kommunikation

Zwischen globalem und lokalem Management findet ein Informationsfluss in zwei Richtungen statt. Die Modulmanager liefern dem Knotenmanager Informationen über den aktuellen Systemzustand. Umgekehrt teilt der Knotenmanager den Modulmanagern mit, welche Reaktionen ausgeführt werden sollen.

- **Erfassung des Systemzustands**: Die Modulmanager führen bereits eine Vorauswertung der gesammelten Daten durch. Dabei bilden sie die verschiedenen Parameterwerte auf eine gemeinsame Domäne ab. So kann zum einen die Kommunikationsschnittstelle zwischen den beiden Verwaltungsebenen sehr schlank gehalten werden, und zum anderen kann der Knotenmanager Messdaten, die aus verschiedenen Domänen stammen, direkt miteinander vergleichen ohne deren Herkunft beachten zu müssen.
- **Reaktionen**: In Gegenrichtung teilt der Knotenmanager seine Entscheidungen den einzelnen Modulmanagern mit. Diese führen dann die Reaktion mit Hilfe ihrer assoziierten Aktoren aus. Ebenso kann das globale Management bestimmte Reaktionen auch direkt ausführen, sofern dies unter dem Blickwinkel des Zeitverhaltens vertretbar ist.

Wie man leicht erkennen kann, sind die beiden Informationsflüsse zumindest teilweise voneinander abhängig: dem Ableiten und Auslösen von Reaktionen muss immer eine vorherige Beobachtung des Systemzustands vorausgehen.

4.2.1 Monitore und Aktoren

Es gibt verschiedene Ansätze, Monitore und Aktoren zu implementieren. Die Art der Implementierung hat wiederum Einfluss auf die Auswahl eines Kommu-

nikationsmodells. Im Folgenden werden deshalb zunächst die unterschiedlichen Implementierungsarten für Monitore und Aktoren diskutiert.

4.2.1.1 Monitore

Zur Implementierung eines Monitors existieren die folgenden Konzepte:

Instrumentierung der Anwendung Der Monitor wird direkt in den Code der Anwendung integriert. Dieser Ansatz eignet sich besonders dann, wenn das Zeitverhalten der Anwendung überwacht werden soll. Dabei ist aber auch das Zeitverhalten des Monitors zu beachten, das sich direkt auf das Zeitverhalten der Anwendung auswirkt.

Separater Dienst Dabei läuft der Monitor in einem eigenen Thread unabhängig von der Hauptanwendung des Knotens. Diese Methode kann etwa bei der Funktionsüberwachung einer Applikation zum Einsatz kommen.

Expliziter Aufruf durch Manager Der Globale Manager fordert Statusinformationen explizit an. Sie werden also nicht wie bei den anderen genannten Methoden selbstständig von den Modulmanagern erstellt. Besonders geeignet ist dieser Ansatz, wenn die Statusinformationen auf einfachen Systemparametern basieren, z.B. das Auslesen der Taktfrequenz oder des Batteriezustands.

Welches Konzept man letztendlich für einen bestimmten Monitor wählt, hängt stark von der Art und Komplexität des Monitors ab. Die genannten Beispiele geben aber eine grobe Richtlinie.

4.2.1.2 Aktoren

Grundsätzlich stehen zur Implementierung von Aktoren die gleichen Prinzipien zur Verfügung wie bei Monitoren.

Instrumentierung der Anwendung Reaktionen werden direkt als Teil der Anwendung ausgeführt. Hierunter fällt etwa das Anpassen bestimmter Programmparameter. Dabei ist der Einfluss der Reaktionen auf das Zeitverhalten der Anwendung zu beachten.

Separater Dienst Hier sind auch aufwändige Reaktionen möglich, da durch geeignete Schedulingverfahren die Beeinflussung anderer Threads nahezu ausgeschlossen werden kann. Auf diesem Weg kann etwa die Verlagerung einer Anwendung auf einen anderen Knoten durchgeführt werden.

Expliziter Aufruf durch Manager Im Gegensatz zu den beiden erstgenannten Prinzipien stellen sich hier keine Synchronisationsprobleme. Der Einfluss auf das Zeitverhalten der Anwendung ist ebenfalls minimal. Allerdings können sich Verzögerungen in der Abarbeitung des globalen Managements ergeben, falls die Ausführungszeit einer Reaktion unbeschränkt ist.

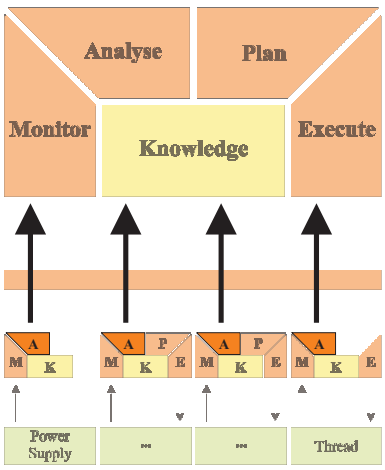
Bei allen Arten von Reaktionen ist zu berücksichtigen, dass sie sich gegebenenfalls mit anderen, parallel ablaufenden Anwendungen synchronisieren müssen. Dies kann das Zeitverhalten der Reaktion entsprechend stark beeinflussen.

Die Wege, Monitore und Aktoren zu implementieren, sind also vielfältig. Wie man sieht, hängt die Auswahl eines bestimmten Wegs immer auch von der Art des Monitors bzw. Aktors ab. Eine Betrachtung grundlegender Kommunikationsmuster wird hier weitere Einsichten liefern.

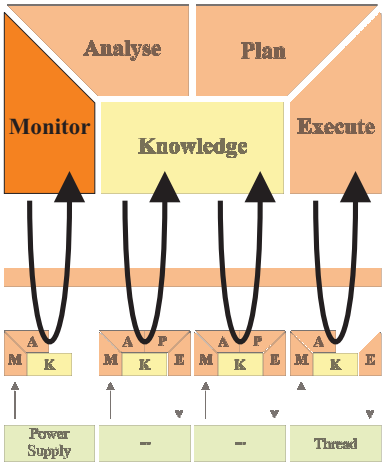
4.2.2 Synchronität und Kommunikationsmuster

Als nächstes stellt sich die Frage, inwieweit Statusmeldung und Reaktion zeitlich zusammenhängen. Eine synchrone Reaktion liegt dann vor, wenn der Empfang einer Statusmeldung in direktem zeitlichen Zusammenhang zur Herleitung einer Reaktion steht. Wenn dagegen keine direkte zeitliche Korrelation zwischen diesen beiden Ereignissen herrscht, liegt eine asynchrone Reaktion vor. Dabei fällt auf, dass die genannten Wege zur Implementierung von Monitoren und Aktoren jeweils eine synchrone oder asynchrone Reaktion begünstigen. Ähnliches gilt auch bei der Betrachtung möglicher Kommunikationsmuster.

Für die Kommunikation zwischen zwei Partnern existieren zwei grundlegende Entwurfsmuster. Zum einen ist dies das ereignisbasierte *Observer-Pattern*, zum anderen das Konzept des *Polling*. Beim Observer-Pattern (Abbildung 4.2(a)) liegt die aktive Rolle bei der Quelle, den Monitoren. Diese benachrichtigen die Senke, den Globalen Manager, über das Vorliegen neuer Statusmeldungen. In ähnlicher Weise arbeitet auch das Publish-Subscribe-Konzept. Ein *Publisher*, im vorliegenden Fall also ein Modulmonitor, veröffentlicht eine Liste von Ereignissen, über die er Bericht erstatten kann. Ein *Subscriber*, hier also die Monitorkomponente des Globalen Managers, wählt aus dieser Liste jene Ereignisse aus, bei deren Auftreten er informiert werden soll und abonniert diese beim Publisher. Der Kommunikationsaufwand wird dadurch minimiert. Allerdings stellt sich die Frage, wann der Globale Manager mit der Herleitung einer Reaktion beginnen soll. Er empfängt die Meldungen der Modulmanager zu unterschiedlichen Zeiten. Es ist vom Aufwand her aber nicht vertretbar, für jede einzelne empfangene Meldung einen Managementzyklus auszulösen. Gleichzeitig ist es aber auch nicht sinnvoll, zu lange mit einer Reaktion zu warten, da dann möglicherweise eine falsche Korrelation der empfangenen Meldungen angenommen wird.



(a) Systemüberwachung gemäß dem Observer-Entwurfsmuster



(b) Systemüberwachung durch Polling

Abbildung 4.2: Entwurfsmuster zum Kommunikationsfluss innerhalb des zweischichtigen Autonomic Managements

Beim Einsatz von *Polling* (Abbildung 4.2(b)) stellt sich diese Frage nicht, da hier immer die Statusnachrichten aller Monitore zu einem Zeitpunkt abgefragt werden. Die Meldungen der Monitore beziehen sich also alle auf den gleichen Zeitpunkt. Aufgrund des gleichzeitigen Empfangs der Statusnachrichten ist eine synchrone Reaktion möglich. Nachteilig wirkt sich hier der hohe Kommunikationsaufwand aus. Der Globale Manager als aktiver Teil der Kommunikation muss jeden Monitor abfragen, unabhängig davon, ob an diesem aktuell überhaupt relevante Meldungen vorliegen.

4.2.3 Hybrides Kommunikationsmodell

Auf Basis der vorausgehenden Diskussion erscheint der Einsatz eines hybriden Kommunikationsmodells sinnvoll. Dieses zielt darauf ab, einerseits die Kommunikation zwischen den Architekturschichten zu minimieren, aber gleichzeitig auch größtmögliche Synchronität bei der Reaktion zu bieten. Dazu werden für den Globalen Manager zwei Modi eingeführt:

- Im Modus **GM async** sind nur die Modulmanager aktiv (Abbildung 4.3 links). Deren Monitorkomponenten senden bei Bedarf Statusnachrichten an den Monitor des Globalen Managers. Dieser akkumuliert die Statusnachrichten zunächst und löst beim Überschreiten einer kritischen Schranke einen Moduswechesl nach *GM sync* aus.

- Der Wechsel in den Modus **GM sync** aktiviert nun auch die restlichen Komponenten Analyse, Planerstellung, Reaktion des Globalen Managers (Abbildung 4.3 rechts). Die Monitorkomponente fragt nun aktiv alle registrierten Modulmanager ab. Auf Basis der gesammelten Daten leitet der Globale Manager eine Reaktion her und wendet diese auf das überwachte System an. Nach Abschluss wechselt der Globale Manager wieder zurück in den Modus *GM async*.

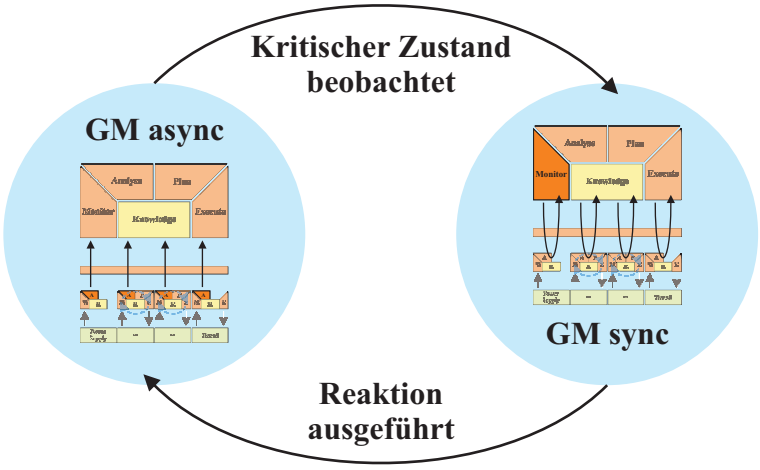


Abbildung 4.3: Zustandsübergänge des hybriden Kommunikationsmodells

Solange also im Produktivsystem alle Vorgänge geregelt ablaufen, ist die zusätzliche Rechenlast durch das globale Management minimal. Erst wenn ein kritischer Zustand eintritt, nimmt das globale Management seine Arbeit auf. Die Definition, wann der Zustand kritisch ist, liegt dabei in der Hand des Benutzers und hängt auch von der Art des gewählten Kommunikationsprotokolls ab. Er kann sich an der Anzahl der akkumulierten Statusmeldungen, aber auch an einer möglichen Gewichtung dieser Meldungen orientieren. Kapitel 5 wird diesen Punkt noch genauer betrachten.

4.2.4 Zusammenarbeit mit einer Middleware

Die vorgestellte Management-Architektur bezieht sich zunächst nur auf einen einzelnen Knoten. Mehrere Knoten innerhalb eines verteilten Systems sind üblicherweise durch eine Middleware miteinander verbunden. Diese kann ihrerseits weitere Techniken bereitstellen, um die Funktionsfähigkeit des gesamten Netzes zu verbessern. In manchen Fällen ist es damit notwendig, dass das globale Autonomic Management der Middleware mit den Managern auf den einzelnen Knoten interagiert.

Eine solche Interaktion ist ohne Erweiterung der zweischichtigen Management-Architektur möglich. Der Knotenmanager stellt bereits eine generische Schnittstelle zur Kommunikation mit den einzelnen Modulmanagern bereit. Das Middleware-Management kann nun selbst einen solchen Modulmanager bereitstellen, welcher in geeigneter Weise Statusinformationen und Reaktionen bereitstellt. Ebenso kann die Middleware von den Modulmanagern Zustandsinformationen abfragen, um daraus eigene Entscheidungen herzuleiten. Dabei muss sie sich aber mit dem Knotenmanager abstimmen.

Eine weitere Schnittstelle für eine Beeinflussung durch die Middleware stellt die Wissensbasis des Knotenmanagers dar. Hier werden üblicherweise gewisse Planungs- und Reaktionsstrategien durch den Benutzer vorgegeben. Aber auch die Middleware kann diese Strategien gemäß ihren eigenen Vorgaben abändern und auf die aktuelle Situation im Netz anpassen.

4.3 Fazit

Der hier vorgestellte Architekturansatz greift die Architekturen des Autonomic und Organic Computing auf. Aufgrund der Trennung in Modul- und Knotenmanager ähnelt sie der verteilten Observer/Controller-Architektur mit übergeordneter Regelschleife (Abbildung 2.6(b), Abschnitt 2.3). Sie verzichtet aber bewusst auf eine Interaktion zwischen den Modulmanagern, da diese zusätzlichen Abhängigkeiten deren allgemeine Einsetzbarkeit reduzieren würde. Die echtzeitfähige Reaktion, die die Observer/Controller-Architektur durch den Action Selector bereitstellt, wird hier durch die Modulmanager erreicht. Anders als in der Observer/Controller-Architektur verfügt der Globale Manager nicht über einen Präprozessor. Stattdessen wird die Vorauswertung der Monitordaten bereits in den Modulmanagern vorgenommen. Dadurch kann das globale Management über eine einheitliche Schnittstelle angesprochen werden.

Man mag einwerfen, dass es zielführender wäre, problemspezifische komplexe Modulmanager zu entwerfen und auf einen globalen Manager zu verzichten. Aus dieser Herangehensweise ergeben sich allerdings verschiedene Probleme etwa für das Zeitverhalten der Anwendungen. Da komplexere Modulmanager zwangsläufig auf einem größeren Parametersatz arbeiten, führt dies auch zu einer höheren Komplexität bei deren Entscheidungsfindung und damit zu einer größeren Laufzeit der Entscheidungsfindung, da jeder zusätzliche Parameter die Dimension des Problems erhöht. Ein anderer Punkt sind die hohen Entwicklungskosten. Die genannten komplexen Modulmanager behandeln Spezialfälle im Hinblick auf ihren Parametersatz. So muss für jede neue Anwendung auch ein neuer Satz Modulmanager entwickelt werden, die Wiederverwendbarkeit der Modulmanager ist hier aufgrund ihrer Spezifität als eher gering einzuschätzen. Im Gegensatz dazu

können die hier geforderten Modulmanager leicht wiederverwendet werden, da sie sich *per definitionem* auf ein in sich abgeschlossenes austauschbares Modul beziehen, z.B. eine einzelne Hardwarekomponente. Das Zeitverhalten einer Anwendung wird hier weit weniger beeinflusst, da die Dimension des Problems (der Parametersatz) kleiner ist. Der Einsatz problemspezifischer Lösungsansätze, wie er etwa von David et al. [12] für das Betriebssystem Choices vorgestellt wurde, ist aber trotzdem nicht ausgeschlossen. Einzige Maßgabe dafür ist, dass sich ein solcher Lösungsansatz mit vertretbarem Aufwand innerhalb eines Modulmanagers umsetzen lässt.

Die Lösung komplexer Probleme erfolgt im Globalen Manager mit Hilfe eines generischen Regelsystems ähnlich der Controller-Komponente der Observer/Controller-Architektur. Die Anwendungsspezifität ergibt sich hier aus der Kombination der Modulmanager sowie den durch den Entwickler vorgegebenen Strategien. Das Verfahren selbst aber wird durch die Art der Anwendung nicht beeinflusst. Einen Ansatz für ein generisches Regelsystem stellt das nächste Kapitel vor.

5 Generisches Management

Das folgende Kapitel befasst sich mit dem Entwurf eines generischen Management-Konzepts für die in Kapitel 4 vorgestellte zweischichtige Management-Architektur. Wie bereits in Abschnitt 4.3 dargelegt wurde, ist auf globaler Ebene die Nutzung eines generischen Algorithmus' zielführend, da dieser unabhängig von der Problemdomäne vielfach eingesetzt werden kann. Der Algorithmus orientiert sich an dem Konzept der *generischen Programmierung* wie etwa dem Einsatz von Templates in C++, die jeweils für eine Klasse von Problemen eine allgemeine Lösung bereitstellen. Diese Lösung ist unabhängig vom konkreten Einsatzgebiet. Dementsprechend stellt der hier entworfene Algorithmus ein allgemeines Werkzeug dar, um zu gegebenen Systemzuständen eine angemessene Reaktion herzuleiten. Das in dieser Arbeit betrachtete Umfeld der eingebetteten Echtzeitsysteme stellt dabei einige zusätzliche Anforderungen an einen solchen Algorithmus. Da eingebettete Echtzeitsysteme im Allgemeinen beschränkt hinsichtlich ihrer Ressourcen sind, muss der Algorithmus entsprechend ressourcenschonend implementiert werden können. Dies betrifft zum einen die Datenbasis, auf der der Algorithmus arbeitet. Für diese ist eine möglichst speichereffiziente Darstellung vonnöten. Ebenso muss der Algorithmus möglichst effizient mit der Rechenzeit umgehen. Er darf parallel ablaufende Echtzeitanwendungen nicht beeinflussen. Zudem soll er nur einen geringen Anteil der zur Verfügung stehenden Rechenzeit beanspruchen. Zuletzt muss sichergestellt sein, dass die hergeleiteten Reaktionen das Verhalten des Systems nicht in unvorhersehbarer und insbesondere nicht in negativer Weise beeinflussen.

5.1 Klassifizierung für eingebettete Echtzeitsysteme

Wie bereits in Abschnitt 2.3.3 erläutert wurde, weisen die dort betrachteten Techniken zur Lösung komplexer Probleme einige entscheidende Nachteile bezüglich ihres Einsatzes in eingebetteten Echtzeitsystemen auf. Learning Classifier Systems können aufgrund des zufallsorientierten genetischen Algorithmus' nicht immer eine geeignete Reaktion garantieren. Der Einsatz automatische Planer hingegen ist im Allgemeinen mit einem hohen Rechenaufwand verbunden. Unter anderen Aspekten eignen sie sich wiederum sehr gut für dieses Einsatzgebiet. Learning Classifier Systems arbeiten mit einer sehr effizienten Zustandsdarstel-

lung. Automatische Planer beachten auch immer die Auswirkungen von Aktionen, so dass sie eine positive Wirkung garantieren können. Der Entwurf einer *dynamischen Klassifizierung für eingebettete Echtzeitsysteme (Dynamic Classification for Embedded Real-Time systems, DCERT)* zielt darauf ab, diese positiven Eigenschaften zu vereinigen und dabei die Nachteile der zugrundeliegenden Techniken zu vermeiden. Ein wichtiges Ziel ist dabei ein möglichst geringer Ressourcenverbrauch bezüglich der Speichernutzung und des Rechenaufwands. Dazu übernimmt DCERT die Zustandsdarstellung der Learning Classifier Systems, welche in einem geringen Speicherverbrauch resultiert und einen effizienten Vergleich von Zuständen erlaubt. In Anlehnung an automatische Planer betrachtet DCERT aber auch genau die Auswirkungen, die eine mögliche Aktion auf das System hat und stellt so sicher, dass es das Systemverhalten nicht negativ beeinflusst.

Die Kommunikation von DCERT mit den Modulmanagern erfolgt über zwei Schnittstellen. *Monitore* (Abschnitt 5.1.1) liefern aktuelle Statusinformationen über die mit ihnen verbundenen Systemkomponenten. Aufgrund dieser Nachrichten wählt DCERT geeignete *Aktoren* (Abschnitt 5.1.2) aus, um auf den aktuellen Systemzustand einzuwirken. Die Kommunikation selbst erfolgt in Form abstrakter Statusnachrichten. Die folgenden Abschnitte erläutern diese Komponenten aus Sicht von DCERT sowie deren Zusammenarbeit bei der Problemlösung.

5.1.1 Monitore

Die Monitore der Modulmanager haben direkten Zugriff auf die Rohwerte der von ihnen überwachten Systemparameter. Sie abstrahieren diese Rohwerte in ein allgemeines Format und leiten diese Informationen bei Bedarf an DCERT weiter. DCERT erkennt in diesen Informationen Missstände und wählt geeignete Reaktionen aus. Das folgende Beispiel 5.1 zeigt einige Möglichkeiten, welche Systemkomponenten überwacht werden können.

Beispiel 5.1 (Überwachte Systemparameter) *In einem eingebetteten System können üblicherweise unter anderem folgende Parameter überwacht werden:*

Prozessor *Der Prozessor eines Systems bietet zwei Parameter, die durch entsprechende Monitore überwacht werden können. Zum einen ist dies die Taktfrequenz. Moderne Prozessoren bieten die Möglichkeit, ihre Taktfrequenz der aktuellen Rechenlast anzupassen. Mit der Taktfrequenz kann üblicherweise auch die Versorgungsspannung verändert werden, wodurch sich erhebliche Energieeinsparungen erzielen lassen.*
Zum anderen kann ein Monitor auch die Auslastung des Prozessors beobachten. Gerade in Echtzeitsystemen muss eine Überlastung des Prozessors vermieden werden, damit die Echtzeitanwendungen ihre Zeitschranken

nicht verpassen. Bei Verwendung eines zeitbasierten Schedulingverfahrens kann die Prozessorauslastung in Form der reservierten Rechenzeit angegeben werden.

Batterie *Auch an der Batterie bzw. Stromversorgung eines eingebetteten Systems können mindestens zwei Parameter überwacht werden. Naheliegend ist hier zunächst die verbleibende Restenergie in der Batterie, die entscheidenden Einfluss auf die längerfristige Funktionsfähigkeit des Systems hat. Wichtig ist auch die aktuelle Leistungsaufnahme, da sie die Lebensdauer der Batterie entscheidend beeinflussen kann. Hierbei ist zu berücksichtigen, dass eine hohe Leistungsaufnahme die aus der Batterie entnehmbare Energiemenge überproportional reduziert. Die Nominalkapazität einer Batterie wird für eine bestimmte Entladerate C angegeben. Venis [63] konnte zeigen, dass sich die entnehmbare Energiemenge für eine Batterie um 10% erhöht, wenn die Batterie stattdessen mit einer Rate von $C/5$ entladen wird.*

Peripherie *Anwendungen sind meistens auf das Vorhandensein von bestimmten Peripheriegeräten angewiesen. Deren Funktionsstatus von Peripheriegeräten kann entweder binär (funktioniert/nicht) oder, sofern sinnvoll, in einem kontinuierlichen Spektrum (Quality of Service, QoS) angegeben werden.*

Software *Auch einzelne Softwarekomponenten können das Systemverhalten beeinflussen. So darf ein Migrationsdienst etwa nur begrenzt Anwendungen starten, um die Funktionsfähigkeit des Knoten nicht einzuschränken.*

Die meisten der Parameter in Beispiel 5.1 nehmen ihre Werte in einem mehr oder weniger kontinuierlichen Spektrum an. Zur Weiterverarbeitung durch DCERT müssen diese Werte stark diskretisiert werden. Im Allgemeinen sind für eine Reaktionsentscheidung nicht die konkreten Werte der Parameter relevant, sondern nur, ob der Parameter gewisse Schwellen über- oder unterschreitet. Dieser Sachverhalt lässt sich mit einfachen booleschen Variablen darstellen.

Definition 5.2 (DCERT-Statusparameter) *Ein* ***Statusparameter*** *wird mit π bezeichnet. Er kann als* ***Statusnachricht*** *die Werte 0 und 1 annehmen ($\pi \in \{0, 1\}$). Die Menge aller Statusparameter innerhalb eines Systems ist $\Pi = \{\pi_0, \pi_1, \pi_2, \ldots \pi_n\}$.*

Die Art und Weise, wie ein Statusparameter π aus den Rohdaten erzeugt wird, ist in den Modulmanagern festgelegt. Im Allgemeinen kann π als wahr/falsch-evaluierbare Aussage aufgefasst werden. Das folgende Beispiel 5.3 verdeutlicht den Einsatz von Statusparametern.

Beispiel 5.3 (DCERT-Statusparameter) *Dieses Beispiel benutzt Monitore für folgende Systemkomponenten*

- *Taktfrequenz*
- *Scheduler*
- *Batterie, Energieverbrauch*
- *Migrationsdienst*

Hieraus lassen sich die folgenden einfachen Statusnachrichten ableiten:

π	*Beschreibung*
f_{max}	*Taktfrequenz ist maximal*
f_{min}	*Taktfrequenz ist minimal*
p_{high}	*viel überschüssige Rechenkapazität*
p_{low}	*zu wenig Rechenkapazität*
b_{high}	*Batterie ist fast voll*
b_{low}	*Batterie ist fast leer*
c_{high}	*Leistungsaufnahme zu hoch*
c_{low}	*Leistungsaufnahme sehr niedrig*
e_a	*Anwendung a kann auf anderen Knoten verschoben werden*
i	*Migrationsanfrage von anderem Knoten liegt vor*

Bei den genannten Parametern fällt auf, dass einige direkt als Beschreibung eines möglichen Missstands aufgefasst werden können, wie etwa die *Auslastung des Prozessors*. Hieraus lässt sich oft direkt auf eine oder mehrere mögliche Reaktionen schließen. Im Fall der Überlastung des Prozessors wäre dies etwa die Erhöhung der Taktfrequenz, aber auch das Verlagern von Applikationen auf weniger stark belastete Rechenknoten. Andere Parameter hingegen können *wertfrei* betrachtet werden, das heißt, aus ihrer Betrachtung allein kann nicht auf einen Missstand geschlossen werden. Ein solcher Parameter ist der Zustand der Taktfrequenz, die innerhalb von der Hardware vorgegebener Schranken verändert werden kann. Aber auch dieser Parameter kann die Reaktionsauswahl beeinflussen. So ist ein Erhöhen der Taktfrequenz nur dann möglich, wenn sie im aktuellen Zustand noch niedriger als ihr Maximalwert ist. In diesem Sinne stellt also der Zustand der Taktfrequenz eine wichtige Vorbedingung für die Aktion *Taktfrequenz erhöhen* dar.

Parameter, die einen zu beseitigenden Missstand darstellen, werden als *Triggerparameter* bezeichnet. Sie geben an, ob das Ausführen bestimmter Aktionen in einem gegebenen Zustand sinnvoll ist, ob also die Aktion eine Zustandsverbesserung nach sich zieht. Sie sind üblicherweise nicht an bestimmte Aktoren gebunden, sondern können durchaus als Auslöser für mehrere verschiedene Aktionen fungieren.

Definition 5.4 (Triggerparameter) *Ein Statusparameter τ, der einen Missstand darstellt, wird als* ***Triggerparameter*** *bezeichnet. Die Menge aller Triggerparameter eines Systems ist $T = \{\tau_0, \tau_1, \ldots \tau_m\}$, $T \subset \Pi$. Zusätzlich muss für alle $\tau_k \in T$ eine Gewichtsfunktion $w(\tau_k)$ mit folgenden Eigenschaften definiert sein:*

- $\tau_k = 0 \quad \Leftrightarrow \quad w(\tau_k) = 0 \quad \forall\, k \leq m$
- $\tau_k = 1 \quad \Leftrightarrow \quad w(\tau_k) \neq 0 \quad \forall\, k \leq m$
- $i \neq k,\, \tau_i \neq 0,\, \tau_k \neq 0 \quad \Leftrightarrow \quad w(\tau_i) < w(\tau_k) \quad \dot{\vee} \quad w(\tau_i) > w(\tau_k) \quad \forall\, i, k < m$

Aufgrund dieser Definition sind Triggerparameter somit geeignet, eine Bewertung des aktuellen Systemzustands durchzuführen. Das folgende Beispiel 5.5 gibt einen kurzen Überblick über die Auswahl von Triggerparametern.

Beispiel 5.5 (Triggerparameter) *Nicht alle der in Beispiel 5.3 definierten Zustände erfordern eine direkte Reaktion. So hat etwa der Zustand der Taktfrequenz zwar Einfluss auf das Systemverhalten, weist aber nicht auf einen möglichen Missstand hin. Dieser wird durch den Zustand p_{low} dargestellt. Dieses Beispiel beschreibt ein batteriebetriebenes Echtzeitsystem, bei dem die Funktionsfähigkeit an höchster Stelle steht. Folgende Zustände werden deshalb als Triggerparameter definiert, das heißt, bei ihrem Auftreten ist eine Reaktion des Managements notwendig:*

Parameter	*Gewicht*	*Erklärung*
$\mathbf{p_{low}}$	*8*	*Sobald zu wenig Rechenzeit zur Verfügung steht, kann die Funktionsfähigkeit des Systems nicht mehr garantiert werden. Dieser Missstand ist daher dringend zu beseitigen.*
$\mathbf{b_{low}}$	*4*	*Bei einem niedrigen Batteriestand kann die Funktionsfähigkeit des Geräts nur noch für einen kurzen Zeitraum aufrecht erhalten werden. Hier müssen geeignete Sicherheitsroutinen sowie gegebenenfalls eine kontrollierte Abschaltung des Geräts eingeleitet werden.*
$\mathbf{c_{high}}$	*2*	*Eine hohe Leistungsaufnahme beeinflusst die Lebensdauer der Batterie und somit auch die Funktionsfähigkeit des Systems negativ.*
$\mathbf{i}$	*1*	*Falls von anderen Knoten eines verteilten Systems Migrationsanfragen vorliegen, so müssen diese beantwortet werden.*

Die Gewichte w der Statusparameter wurden hier in exponentieller Ordnung belegt. Hieraus ergibt sich ein Vorteil für die effiziente Implementierung mittels

Bitstrings, wie in Abschnitt 5.2.1 genauer besprochen. Die Ordnung der Triggerparameter ist implizit durch deren Gewichte gegeben.

Aufgrund ihrer Definition stellen Triggerparameter eine kanonische Schwelle für den Moduswechsel der internen Kommunikation in Abschnitt 4.2.3 dar. Das Auftreten einer Triggernachricht kann den globalen Manager aufwecken, so dass dieser aktiv Statusnachrichten von allen Monitoren sammelt. Dies ist sinnvoll, da beim Auftreten einer Triggernachricht ein negativer Einfluss auf das Gesamtsystem zu befürchten ist.

Zur weiteren Arbeit mit Triggerparametern wird noch die folgende Definition benötigt:

Definition 5.6 (Gewicht einer Triggermenge) *Das Gewicht einer Menge aus Triggerzuständen $U \subset T$ berechnet sich als Summe der Einzelgewichte ihrer Zustände:*

$$w(U) = \sum_{\tau \in U} w(\tau)$$

5.1.2 Aktoren

Aktoren stellen mögliche Reaktionen auf bestimmte Systemzustände bereit. Die Auswahl eines Aktors erfolgt durch DCERT auf Basis des aktuellen Systemzustands, sowie den folgenden Eigenschaften von Aktoren. Die Auswahl eines Aktors zur Reaktion muss sich an mehreren Aspekten orientieren. Zunächst muss seine Ausführung im aktuellen Zustand möglich und sinnvoll sein. Außerdem muss die Reaktion zu angemessenen Kosten erfolgen können. Dazu werden Aktoren wie folgt definiert:

Definition 5.7 (Aktor) *Ein Aktor A ist ein 4-Tupel $A = (a, C, U, s)$ mit folgenden Elementen:*

- *a bezeichnet die Aktion, die durch A ausgeführt werden kann;*
- *$C \subset \Pi$ bezeichnet die notwendigen Voraussetzungen für die Ausführung von a, wie sie bereits unter 5.1.1 erwähnt wurden;*
- *die Triggermenge $U \subset T$ fasst all jene Missstände in Form von Triggerparametern zusammen, die a auslösen können und die durch a beeinflusst werden (siehe ebenfalls 5.1.1); zudem soll gelten $C \cap U = \emptyset$, ein Trigger darf also nicht gleichzeitig Voraussetzung sein;*
- *Der Kostenskalawert s stellt ein abstraktes Maß für die Komplexität und tatsächlichen Kosten von a dar.*

Die Menge C der Voraussetzungen eines Aktors zeigt auf, wann seine Ausführung möglich ist. Die Elemente U und s eines Aktors bilden sein Verhalten und seine Wirkung auf das System ab. Die Triggermenge U umfasst jene Statusparameter, bei deren Auftreten die Ausführung von a sinnvoll ist. Implizit bedeutet dies, dass der Aktor A versucht, die vorhandenen Triggerzustände aus U zu beseitigen. Der Kostenskalawert s wird benötigt, wenn DCERT eine Auswahl zwischen zwei Aktoren mit ähnlichen Triggermengen treffen muss. Seine Behandlung wird in Abschnitt 5.1.3 erläutert.

Beispiel 5.8 (Aktoren) *Dieses Beispiel stellt zwei Aktoren zur Veränderung der Taktfrequenz vor. Dazu werden die Statusparameter aus Beispiel 5.5 verwendet.*

	`IncFreqActor`	`DecFreqActor`
a	*Erhöhen der Taktfrequenz f um Δf*	*Senken der Taktfrequenz f um Δf*
C	$\{\neg f_{max}, \neg b_{low}\}$ *f kann erhöht werden ($f < f_{max}$)*	$\{\neg f_{min}\}$ *f kann gesenkt werden ($f > f_{min}$)*
U	$\{p_{low}\}$ *Anwendung bei niedriger Rechenleistung, Energieaufnahme erhöht sich*	$\{c_{high}, b_{low}, p_{high}\}$ *Anwendung bei zu hoher Energieaufnahme, verfügbare Rechenleistung sinkt*
s	$s_{inc} = 1$	$s_{dec} = 1$

Der Kostenskalawert der beiden Aktoren ist sehr niedrig, da es sich bei der Veränderung der Taktfrequenz typischerweise um eine sehr einfache Aktion handelt.

Weitere zwei Aktoren können durch den dynamischen Linker des Betriebssystems bereitgestellt werden:

	`ImmigrationActor`	`EmigrationActor`
a	*Empfang und Ausführung eines Dienstes auf dem Knoten*	*Verlagerung eines Dienstes auf einen anderen Knoten*
C	$\{\neg p_{low} \vee (b_{high}, f_{max})\}$ *Ausreichend Rechenzeit vorhanden oder erzeugbar*	$\{e_a\}$ *ein verlagerbarer Dienst läuft auf diesem Knoten*
U	$\{i\}$ *Ein Dienst wartet auf seine Einlagerung*	$\{p_{low}\}$ *Zu hohe Knotenlast*
s	$s_{rec} = 10$	$s_{send} = 10$

Das Empfangen oder Versenden einer Anwendung ist mit erheblichem Aufwand verbunden, deshalb wird diesen beiden Aktionen ein deutlich höherer Kostenskalawert zugewiesen.

Die Festlegung der einzelnen Elemente der Aktoren erfolgt durch den Entwickler. Voraussetzungen C_A und Triggermenge U_A lassen sich im Allgemeinen relativ leicht aus der Aktion herleiten. Bei der Festlegung der Kostenskalawerte ist der Entwickler dagegen zunächst keinerlei Einschränkungen unterworfen. Hier ist einzig darauf zu achten, dass die Bewertungen unterschiedlicher Aktionen untereinander in einem angemessenen Verhältnis stehen. Insofern ist eine Festlegung der Kostenskalawerte erst bei der Integration eines Systems sinnvoll.

5.1.3 Entscheidungsfindung

Wie in Abschnitt 4.2.3 bereits dargestellt wurde, ist der globale Manager im Normalbetrieb inaktiv. Das Auftreten eines Triggerzustands löst die Entscheidungsfindung aus. Der globale Manager wechselt hierauf in den synchronen Modus und ruft aktiv die Statusnachrichten aller registrierten Modulmanager ab. Mit Hilfe der im Folgenden dargestellten Entscheidungslogik findet der globale Manager eine Reaktion unter den verfügbaren Aktoren, um vorherrschende Missstände zu beseitigen. Dabei beachtet er gegebenenfalls weitere Rahmenbedingungen wie etwa die Minimierung der Kosten oder der negativen Seiteneffekte, aber auch die Dauer, über die ein Missstand bereits anhält. Grundlage für all diese Entscheidungen ist der aktuelle Weltzustand:

Definition 5.9 (Weltzustand) *$S_t \subset \Pi$ heißt **Weltzustand** zum Zeitpunkt t. S_t enthält die Statusmeldungen aller Monitore, die diese zum Zeitpunkt t melden.*

Beispiel 5.10 (Weltzustand) *Seien Π und T wie in den Beispielen 5.3 und 5.5. $S_t = \{f_{max}, p_{high}, c_{high}, i\}$ beschreibt einen Zustand, in dem*

- *die Taktfrequenz maximal ist,*
- *der Prozessor über viel freie Rechenzeit verfügt,*
- *die Leistungsaufnahme des Prozessors sehr hoch ist, und*
- *wenigstens eine Migrationsanfrage von einem anderen Knoten vorliegt.*

Falls der aktuelle Weltzustand kritisch ist, also wenigstens einen Triggerparameter enthält, so ist es die Aufgabe von DCERT, eine Reaktion auf diesen Zustand zu finden. Dazu muss es die Menge aller registrierten Aktoren schrittweise immer weiter einschränken. Im ersten Schritt bestimmt DCERT eine Grundmenge von Aktoren, deren Triggermengen vom aktuellen Weltzustand getroffen sind und deren Vorbedingungen erfüllt sind:

Definition 5.11 (Verfügbare und ausgelöste Aktoren) *Sei $\mathcal{A}$ die Menge aller Aktoren eines Systems. Ein Aktor $A \in \mathcal{A}$ ist im Weltzustand S_t* ***verfügbar****, wenn all seine Vorbedingungen erfüllt sind:*

$$C_A \subseteq S_t$$

Ein verfügbarer Aktor A ist ***ausgelöst****, wenn zusätzlich gilt*

$$U_A \cap S_t \neq \emptyset \ ,$$

er also einen vorherrschenden Missstand beseitigt.

Die Menge an verfügbaren, ausgelösten Aktoren wird im Folgenden mit $\mathcal{C}_0 \subset \mathcal{A}$ bezeichnet. Diese kann noch weiter eingeschränkt werden, um schließlich eine Reaktionsmenge zu erhalten. Solche Einschränkungen sind aus mehreren Gründen nötig. So kann es vorkommen, dass mehrere Aktoren aus $\mathcal{C}_0$ auf denselben Triggerzustand auslösen. In diesem Fall soll vermieden werden, dass dieser Triggerzustand mehrfach behandelt wird, oder die Aktoren sich gegenseitig blockieren. Ebenso ist es möglich, dass sich zwei Aktoren in ihren Vorbedingungen überlappen, und gleichzeitig die Ausführung eines Aktors diese Vorbedingung beseitigt. Die Ausführung des zweiten Aktors wäre damit nicht mehr möglich. DCERT verfolgt daher das Ziel, in jedem Durchlauf die Grundmenge $\mathcal{C}_0$ auf nur einen einzelnen Aktor zur Reaktion einzuschränken. Dadurch vermeidet es die genannten Probleme, und jede Reaktion passt exakt auf den aktuellen Zustand.

Für die Einschränkung bieten sich mehrere Strategien an. Der Begriff Aktoren bezieht sich im Folgenden auf die verfügbaren, ausgelösten Aktoren aus $\mathcal{C}_0$ mit nicht-disjunkten Triggermengen.

Effekt Wähle den Aktor, der auf den wichtigsten Trigger anspricht

- $\oplus$ schlimmster Missstand wird am ehesten behandelt
- $\ominus$ unwichtige Probleme werden möglicherweise nie behandelt

Kosten Wähle den Aktor, dessen Ausführung die geringsten Kosten verursacht

- $\oplus$ schnelle Reaktion
- $\ominus$ möglicherweise wirkungslos, teuerere Aktion wäre besser

Dauer Wähle den Aktor, der auf den am längsten anhaltenden Triggerzustand anspricht

- $\oplus$ auch lang anhaltende, aber unwichtige Probleme werden behandelt
- $\ominus$ Behandlung wichtiger, aber neuer Probleme wird verzögert

Wie man sieht, haben alle der genannten Strategien ihre Vor- und Nachteile. Ziel jeder Aktion muss es aber sein, ein möglichst funktionsfähiges System zu erhalten. Aus diesem Grund soll der Effekt eines Aktors als wichtigstes Kriterium angesehen werden:

Definition 5.12 (Filterung auf Trigger) *Seien mehrere Aktoren ausgelöst durch $\tau \in S_t$. Zur Reaktion soll jener Aktor kommen, der bezüglich des S_t die größte Verbesserung verspricht. Die Verbesserung $b_{S_t}(A)$ eines Aktors A bezüglich S_t berechnet sich als:*

$$b_{S_t}(A) = w(S_t \cap U_A)$$

Es wird derjenige Aktor A ausgewählt, für den $b_{S_T}(A)$ maximal ist.

Trotzdem ist es möglich, dass nach diesem Filterschritt immer noch mehrere Aktoren zur Behandlung desselben Triggerzustands zur Verfügung stehen. Dies ist etwa der Fall bei den beiden Aktoren `IncFreqActor` und `EmigrationActor` aus Beispiel 5.8. Beide versprechen, die verfügbare Rechenzeit zu erhöhen. Hier bietet es sich nun an, die beiden anderen Strategien *Kosten* und *Dauer* zu einer weiteren Filterregel zu kombinieren. Dabei wird die Dauer, über die ein Triggerzustand bereits anhält, als eine Art Guthaben genutzt, um höhere Kosten eines Aktors zu gerechtfertigen.

Definition 5.13 (Dauerzähler) *Definiere für jeden Trigger τ einen Dauerzähler c_τ. Anfangs gelte $c_\tau = 0$. Bei jedem Klassifikationsdurchlauf wird c_τ wie folgt verändert:*

- $c_\tau \in S_t$*:* $c_\tau := c_\tau + 1$
- $c_\tau \notin S_t$*:* $c_\tau := c_\tau - 1$

Wiederholtes Auftreten eines Triggers führt also zur Erhöhung seines Dauerzählers. Wird der Triggerzustand durch eine geeignete Reaktion beseitigt, so sinkt der Dauerzähler wieder. Durch das nur allmähliche Absinken des Zählers wird erreicht, dass auch auch bei kurzzeitigem Nicht-Auftreten eines Missstands das Problem weiterhin als dringend betrachtet wird. Mit Hilfe des Dauerzählers kann nun folgend Auswahlregel definiert werden:

Definition 5.14 (Filterung auf Kosten und Dauer) *Seien A_1, A_2 ausgelöst durch $\tau \in S_t$. Zur Reaktion ausgewählt werden soll dann A_x mit*

$$s_{A_x} = \max_{k \in \{1,2\}} \{s_{A_k} \mid s_{A_k} \leq c_\tau\}$$

Kann auch dieser Filterschritt keine Entscheidung zwischen mehreren Aktoren treffen, so sind die Aktoren aus Sicht von DCERT gleichwertig. In diesem Fall kann eine zufällige Auswahl getroffen werden.

5.1.4 Interaktion mit dem Benutzer

Bezüglich eines eingebetteten Systems mit DCERT kann man zwei Arten von Benutzern unterscheiden. Zum einen steht hier der Entwickler des eingebetteten Systems als direkter Nutzer von DCERT, zum anderen gibt es den Endnutzer, der das fertige System benutzt. Die beiden folgenden Abschnitte zeigen, inwieweit diese Benutzer mit DCERT interagieren können oder müssen.

Der Entwickler als Benutzer Der vorgestellte Entwurf lässt dem Entwickler viele Freiheiten, um das Verhalten von DCERT zu beeinflussen. Im einzelnen sind dies:

- Angabe einer Zustandssemantik auf den Triggerparametern mit entsprechender Ordnung und Gewichten;
- Nutzung eigener Kostenfunktionen zur Festlegung der Kostenskalawerte s_A der einzelnen Aktoren;
- eigene Wahl des Schwellwertes für den Moduswechsel der Kommunikation (4.2.3), eingeschränkt durch Trennung in einfache Parameter und Triggerparameter.

Insbesondere bei der Zustandssemantik und deren Ordnung und Gewichtung ist eine Änderung im laufenden Betrieb vorgesehen. Damit kann nicht nur das überwachte System, sondern auch das Verhalten der Überwachungseinheit selbst an veränderte Umgebungsbedingungen angepasst werden. Ein Satz aus den Parametern Zustandssemantik und Ordnung/Gewichte wird im Folgenden als *Systemstrategie* bzw. *Strategiedefinition* bezeichnet.

Der Endnutzer Im Idealfall nimmt der Endnutzer das Vorhandensein eines Autonomic Managements gar nicht wahr, sondern hat zu jedem Zeitpunkt ein voll funktionsfähiges System vor sich. Ein direktes Eingreifen in die Managementfunktionalität erscheint auch wenig zielführend, da der Endnutzer im Allgemeinen nicht über das dafür notwendige Fachwissen über die Interna des Systems verfügt. Stattdessen können aber je nach Anwendungsbereich verschiedene Sätze mit Systemstrategien (siehe oben) durch den Entwickler bereitgestellt werden, welche entweder automatisch oder auch auf explizite Anforderung durch den Endnutzer eingesetzt werden.

5.2 Implementierung von DCERT

Die Zielplattform dieser Arbeit ist ein eingebettetes System, das unter Echtzeitbedingungen arbeitet. Daraus ergeben sich einige Anforderungen an die Implementierung des Managements. Um das Zeitverhalten der Echtzeitanwendung nicht zu beeinflussen, bietet es sich an, das Management als isolierten Helper Thread auf einem mehrfädigen Prozessor laufen zu lassen. In einem herkömmlichen einfädigen Prozessor ist dagegen ein Betriebssystem mit zeitgetriebenem Scheduling notwendig. Die zur Verfügung stehende Rechenzeit ist aber in beiden Fällen stark begrenzt. Die Implementierung muss aber auch mit den anderen Systemressourcen sehr effizient umgehen. Da eingebettete Systeme üblicherweise auch nur über wenig Speicher verfügen, wird insbesondere für die Laufzeitdaten eine speichereffiziente Darstellung benötigt. Ebenso steht auch nur begrenzter Speicherplatz für Programmcode zur Verfügung. Somit ist eine äußerst effiziente Implementierung gefragt. Der folgende Abschnitt beschreibt einen möglichen Weg dazu.

5.2.1 Konzept für Statusparameter

Statusparameter nach Definition 5.2 können, wie in Tabelle 5.1 dargestellt, durch je einen einzelnen Bitwert repräsentiert werden.

Tabelle 5.1: 1-Bit Repräsentation von Statusparametern

Bitwert	Parameter
0	π^0
1	π^1

Ein Weltzustand S_t kann damit als Bitstring dargestellt werden. Den einzelnen Statusparametern sind feste Positionen innerhalb dieses Bitstrings zugeordnet. Abbildung 5.1 zeigt die Nutzung eines 8-Bit-Wortes zur Speicherung von acht einfachen Zustandsparametern. Jedem Parameter ist dabei eine Bitposition zugeordnet, die den Wert 0 oder 1 annehmen kann.

Bitposition p_b	7	6	5	4	3	2	1	0
	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Parameterposition p_p	7	6	5	4	3	2	1	0
Parameter	π_7	π_6	π_5	π_4	π_3	π_2	π_1	π_0

Abbildung 5.1: Bit- und Parameterpositionen am Beispiel eines 8-Bit-Wortes, einfache Statusparameter

Für Triggerparameter benötigt DCERT zusätzlich noch Gewichte, die die Relevanz der einzelnen Parameter für das System bewerten. Die Darstellung von Parametern über ihre Position innerhalb eines Bitstrings bringt eine mögliche Gewichtung gleich mit sich. Einem Triggerparameter τ an Bitposition p_τ ist dabei das Gewicht $w_\tau = 2^{p_\tau}$ zugeordnet. Das Gewicht einer Triggermenge erhält man, indem man alle Nicht-Triggerparameter aus dem Zustands-Bitstring ausmaskiert und diesen dann als Ganzzahl interpretiert. Das folgende Beispiel 5.15 erläutert diese Vorgehensweise.

Beispiel 5.15 (Implementierung von Statusparametern) *Dieses Beispiel zeigt eine mögliche Umsetzung der Status- und Triggerparameter aus den Beispielen 5.3 und 5.5. Insgesamt existieren zehn Statusparameter. Gewichte sind aber nur denjenigen Statusparametern zugeordnet, die auch Triggerparameter sind.*

Bitposition	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Parameter	p_{low}	b_{low}	c_{high}	i	e_a	c_{low}	p_{high}	b_{high}	f_{min}	f_{max}
Triggergewicht	*512*	*256*	*128*	*64*	*./.*	*./.*	*./.*	*./.*	*./.*	*./.*

5.2.2 Aktoren

Die Voraussetzungs- und Triggermengen C und U von Aktoren werden in einer ähnlichen Weise mit Bitstrings dargestellt. Die Darstellung der Triggermenge besteht wiederum aus einem einzelnen Bitstring. Ein gesetztes Bit bedeutet hier, dass der zugehörige Triggerzustand die Aktion auslösen kann. Die Darstellung der Voraussetzungsmenge benötigt bei Nutzung von Bitstrings für jeden Parameter drei mögliche Werte. Zusätzlich zu den Werten 0 und 1 ist ein *Don't-care*-Symbol nötig, üblicherweise als "#" dargestellt. Dieses gibt an, dass der Parameter an der entsprechenden Stelle für den Aktor irrelevant ist. Eine Voraussetzungsmenge wird daher als Kombination aus zwei Bitstrings dargestellt. Eine *Relevanzmaske* gibt an, welche Statusparameter für den Aktor überhaupt relevant sind. Die *Parametermenge* enthält dann die eigentlichen Werte der Statusparameter. Tabelle 5.2 schlüsselt die Kombination aus Relevanzmaske und Parametermenge sowie deren Bedeutung auf.

Tabelle 5.2: Bitrepräsentation der Voraussetzungsmenge von Aktoren

Relevanzbit	Parameterbit	Bedeutung
1	0	π^0
	1	π^1
0	*	# (π irrelevant)

Das folgende Beispiel führt die vorhergehenden Beispiele fort und zeigt, wie die Parametermengen von Aktoren implementiert werden können.

Beispiel 5.16 (Implementierung von Aktoren) *Die Zustandsmengen C und U der Aktoren aus Beispiel 5.8 haben mit der Belegung aus Beispiel 5.15 folgende Gestalt:*

Erhöhen der Taktfrequenz

`IncFreqActor`: $C = \{\neg b_{low}, \neg f_{max}\}, U = \{p_{low}\}$

***Voraussetzungsmenge** C:*

Parameter	p_{low}	b_{low}	c_{high}	i	e_a	c_{low}	p_{high}	b_{high}	f_{min}	f_{max}
Relevanzmaske	*0*	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*1*
Parametermenge	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
Bedeutung	#	*0*	#	#	#	#	#	#	#	*0*

***Triggermenge** U:*

Parameter	p_{low}	b_{low}	c_{high}	i	e_a	c_{low}	p_{high}	b_{high}	f_{min}	f_{max}
Triggermenge	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*

Senken der Taktfrequenz

`DecFreqActor`: $C = \{\neg f_{min}\}, U = \{c_{high}, b_{low}, p_{high}\}$

***Voraussetzungsmenge** C:*

Parameter	p_{low}	b_{low}	c_{high}	i	e_a	c_{low}	p_{high}	b_{high}	f_{min}	f_{max}
Relevanzmaske	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*0*
Parametermenge	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
Bedeutung	#	#	#	#	#	#	#	#	*0*	#

***Triggermenge** U:*

Parameter	p_{low}	b_{low}	c_{high}	i	e_a	c_{low}	p_{high}	b_{high}	f_{min}	f_{max}
Triggermenge	*0*	*1*	*1*	*0*	*0*	*0*	*1*	*0*	*0*	*0*

Verlagerung eines Dienstes auf einen anderen Knoten

`EmigrationActor`: $C = \{e_a\}, U = \{p_{low}\}$

***Voraussetzungsmenge** C:*

Parameter	p_{low}	b_{low}	c_{high}	i	e_a	c_{low}	p_{high}	b_{high}	f_{min}	f_{max}
Relevanzmaske	*0*	*0*	*0*	*0*	*1*	*0*	*0*	*0*	*0*	*0*
Parametermenge	*0*	*0*	*0*	*0*	*1*	*0*	*0*	*0*	*0*	*0*
Bedeutung	#	#	#	#	*1*	#	#	#	#	#

***Triggermenge** U:*

Parameter	p_{low}	b_{low}	c_{high}	i	e_a	c_{low}	p_{high}	b_{high}	f_{min}	f_{max}
Triggermenge	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*

Übernahme eines Dienstes von einem anderen Knoten
`ImmigrationActor`: $C = \{\neg p_{low} \vee (b_{high}, f_{max})\}, U = \{i\}$

Die Darstellung der Voraussetzungsmenge eines Aktors lässt keine Disjunktion von Statusparametern zu. Aus diesem Grund benötigt dieser Aktor zwei Voraussetzungsmengen $C_1 = \{\neg p_{low}\}$ *und* $C_2 = \{b_{high}, \neg f_{max}\}$

Voraussetzungsmenge C_1***:***

Parameter	p_{low}	b_{low}	c_{high}	i	e_a	c_{low}	p_{high}	b_{high}	f_{min}	f_{max}
Relevanzmaske	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
Parametermenge	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
Bedeutung	*0*	#	#	#	#	#	#	#	#	#

Voraussetzungsmenge C_2***:***

Parameter	p_{low}	b_{low}	c_{high}	i	e_a	c_{low}	p_{high}	b_{high}	f_{min}	f_{max}
Relevanzmaske	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*0*	*1*
Parametermenge	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*0*	*0*
Bedeutung	#	#	#	#	#	#	#	*1*	#	*0*

Triggermenge U***:***

Parameter	p_{low}	b_{low}	c_{high}	i	e_a	c_{low}	p_{high}	b_{high}	f_{min}	f_{max}
Triggermenge	*0*	*0*	*0*	*1*	*0*	*0*	*0*	*0*	*0*	*0*

5.2.3 Klassifizierung

Bei den Mengenvergleichen, die zur Klassifizierung notwendig sind, handelt es sich überwiegend um die Bildung von Schnittmengen über den Bitstrings. Bei Begrenzung des Parametersatzes auf eine maximale Anzahl von Statusparametern kann ein solcher Bitstring durch eine Ganzzahlvariable dargestellt werden. Für diesen Datentyp stellen heutige Mikroprozessoren bitweise Operationen wie `AND` und `OR` zur Verfügung, mit denen die Mengenvergleiche implementiert werden können.

Beispiel 5.17 (Durchführung der Klassifizierung) *Gegeben sei folgender Weltzustand:* $S = \{p_{low}, b_{high}, e_a\}$*:*

Auf diesem Zustand aufbauend werden nun Aktoren nach folgenden Regeln ausgewählt:

1. *verfügbare Aktoren mit* $C_A \subseteq S$ *(Definition 5.11) und*
2. *ausgelöste Aktoren* $U_A^- \cap S \neq 0$ *(Definition 5.11)*

Parameter		p_{low}	b_{low}	c_{high}	i	e_a	c_{low}	p_{high}	b_{high}	f_{min}	f_{max}
Weltzustand		***1***	***0***	***0***	***0***	***1***	***0***	***0***	***1***	***0***	***0***
IncFreqAct.	C	#	***0***	#	#	#	#	#	#	#	***0***
	U	***1***	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
DecFreqAct.	C	#	#	#	#	#	#	#	#	*0*	#
	U	*0*	*1*	*1*	*0*	*0*	*0*	*1*	*0*	*0*	*0*
Immig.Act.	C_1	*0*	#	#	#	#	#	#	#	#	#
	C_2	#	#	#	#	#	#	#	*1*	#	*0*
	U	*0*	*0*	*0*	*1*	*0*	*0*	*0*	*0*	*0*	*0*
Emig.Act.	C	#	#	#	#	***1***	#	#	#	#	#
	U	***1***	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*

Im vorliegenden Beispiel sind also zwei Aktoren verfügbar und ausgelöst. Die Aktoren `IncFreqActor` *und* `EmigrationActor` *versprechen, den Systemzustand zu verbessern. In den nächsten Schritten muss DCERT nun einen dieser beiden Aktoren zur Reaktion auswählen.*

3. *Triggerfilterung* $\max b_{s_t}(A) = -w(S_T \cap U_A)$ *(Definition 5.12): Die Filterung auf Triggerparameter und die Verbesserung durch eine Aktion bringt hier auch keine weitere Einschränkung der Reaktionsmenge, da hier beide Aktionen auf den Trigger* p_{low} *ansprechen (unterstrichen in obiger Tabelle).*

4. *Kosten & Dauer (Definition 5.13): Die letzte Filtermethode wird in diesem Beispiel zu einer eindeutigen Reaktion führen. Solange der Dauerzähler zum Trigger* p_{low} *sehr niedrig ist, wird DCERT den Aktor* `IncFreqActor` *auswählen, da diese vergleichsweise billig ist (Beispiel 5.8). Steigt der Dauerzähler für diesen Zustand aber weit genug an (das Erhöhen der Taktfrequenz bringt also keine Verbesserung), so kann DCERT auch den Aktor* `EmigrationActor` *auswählen, um anderweitig Rechenkapazität auf dem Knoten freizugeben.*

5.3 Diskussion

DCERT ist so entworfen, dass es sich auf einer Vielzahl heutiger Microcontroller mit geringem Aufwand implementieren lässt. Systemzustände können durch Ganzzahlvariablen dargestellt werden, über die jeder Microcontroller verfügt. Die notwendigen bitweisen Operationen und Vergleiche sind auf den üblichen Plattformen ebenfalls verfügbar. Durch Nutzung solcher intrinsischer Funktionen wie etwa der bitweisen `AND`-Operation ist der Klassifizierungsprozess auch bezüglich seiner Laufzeit sehr effizient. Die Darstellung der Zustände als Ganzzahlvariablen führt auch zu einer geringen Nutzung des Speichers. Dies ist insbesondere bei einer größeren Anzahl von Aktoren wichtig, da diese persistent im System ge-

speichert sein müssen. Bei Nutzung einer heutzutage üblichen 32-Bit-Architektur kann sich der Speicherbedarf eines Aktors wie folgt zusammensetzen:

- Funktionszeiger a auf die Aktion: 4 Bytes
- Relevanzmaske und Parametermenge der Voraussetzung C: 2×4 Bytes
- Triggermenge U: 4 Bytes
- Kostenskalawert s: 4 Bytes

Insgesamt würde ein Aktor damit 20 Bytes belegen. Der Speicher für den Code der Aktion bleibt hier außer Acht, da dieser von der Art und Implementierung der Aktion abhängt. Ohnehin müsste dieser Speicher auch bei Nutzung eines anderen Managementsystems anstatt von DCERT belegt werden.

Abgesehen von den genannten bitweisen Operationen stellt DCERT keine weitergehenden Anforderungen an die unterliegende Hardware. In einfädigen Prozessoren kann es als eigener periodischer Task laufen, für den die Hauptanwendung regelmäßig unterbrochen wird. Diese Unterbrechungen beeinflussen das Zeitverhalten der Hauptanwendung und müssen bei deren WCET-Analyse berücksichtigt werden. Der Vorteil beim Einsatz eines mehrfädigen Prozessors wie dem CarCore liegt darin, dass DCERT hier im Zeitschatten der Hauptanwendung ablaufen kann. Deren Zeitverhalten wird dadurch nicht beeinflusst.

Die Art der Statusparameter muss bereits beim Entwurf eines Systems festgelegt werden. Für eine effiziente Implementierung empfiehlt es sich, hier den Parametern auch bereits feste Bitpositionen innerhalb der verwendeten Variablen zuzuweisen. Um eine noch größere Flexibilität des Gesamtsystems zu erzielen, ist es aber auch möglich, die Bitpositionen erst zur Laufzeit festzulegen und unter Umständen auch zu verändern. Damit wird es etwa möglich, Rechenleistung und Energieeffizienz situationsabhängig zu gewichten. Solange ein solches Gerät nur von einer Batterie mit Energie versorgt wird, soll der energieeffiziente Betrieb an oberster Stelle stehen. Ist es hingegen an ein Stromnetz angeschlossen, kann DCERT umkonfiguriert werden, wodurch dann ein größeres Gewicht auf der verfügbaren Rechenleistung läge. Der Preis hierfür ist zum einen allerdings ein deutlich größerer Speicherbedarf für die Aktoren, da die Voraussetzungs- und Triggermengen nun nicht mehr direkt angegeben werden können, sondern deren Elemente über symbolische Konstanten referenziert werden müssten. Nach jeder Rekonfiguration muss DCERT dann die neuen Bitstrings der Voraussetzungs- und Triggermengen bestimmen. Die Klassifikation selbst ist von dieser Änderung also nicht betroffen.

DCERT stellt eine Technik bereit, mit der ein System flexibel zur Laufzeit auf Missstände reagieren kann. Es setzt allerdings voraus, dass sich der Entwickler beim Entwurf des Systems mit den Überwachungs- und Reaktionsmöglich-

keiten auseinandersetzt, die das System bietet. Insbesondere die Definition der Voraussetzungs- und Triggermengen der Aktoren haben entscheidenden Einfluss auf das Verhalten von DCERT zur Laufzeit. DCERT selbst trifft nur die Entscheidung, *welche* Aktion im aktuellen Zustand ausgeführt werden soll, aber nicht *wie* diese Aktion ausgeführt wird. Insofern können einzelnen Aktoren auf entsprechend niedrigerer Ebene noch weitergehende Entscheidungen treffen.

DCERT ist nicht in der Lage, neue Zusammenhänge herzuleiten, etwa wenn sich Aktoren noch auf andere als die angegeben Voraussetzungs- und Triggerparameter auswirken. Auf eine solche Lernfunktionalität verzichtet DCERT, da hierbei immer auch das Risiko von Fehlentscheidungen bestünde, was in harten Echtzeitsystemen mitunter katastrophale Folgen hätte.

Eine Middleware wie CARISMA [40, 41] kann mit DCERT zusammenarbeiten, indem sie eigene Statusparameter, Monitore und Aktoren definiert. Über die Monitore stellt die Middleware dann Informationen über relevante Zustände in dem verteilten System zur Verfügung. DCERT kann dann entscheiden, entsprechende Middleware-Aktoren aufzurufen, die den lokalen Knoten und seine Rolle in dem verteilten System beeinflussen.

6 Beispielhafte Modulmanager für die Zusammenarbeit mit DCERT

DCERT ist in der Lage, einzelne problemspezifische, aber voneinander unabhängige Managementkomponenten miteinander zu verknüpfen. Es ist dabei vollständig auf diese Modulmanager angewiesen, da es selbst keinerlei direkten Einfluss auf das Produktivsystem nehmen kann. Dieses Kapitel befasst sich mit dem Entwurf einiger beispielhafter Modulmanager, mit denen ein Autonomic Management wie in Abbildung 6.1 aufgebaut werden kann. Das Autonomic Management wird von DCERT und mehreren Modulmanagern gebildet. Die Modulmanager greifen direkt auf das Produktivsystem (*System under Observation/Control*) zu. Bei Problemen, deren Lösung mehr Informationen als nur die eines Moduls berücksichtigen muss, interagieren sie mit DCERT. Dieses hat als übergelagerte Komponente Zugriff auf die Informationen aller Modulmanager und kann so auch komplexere Probleme lösen.

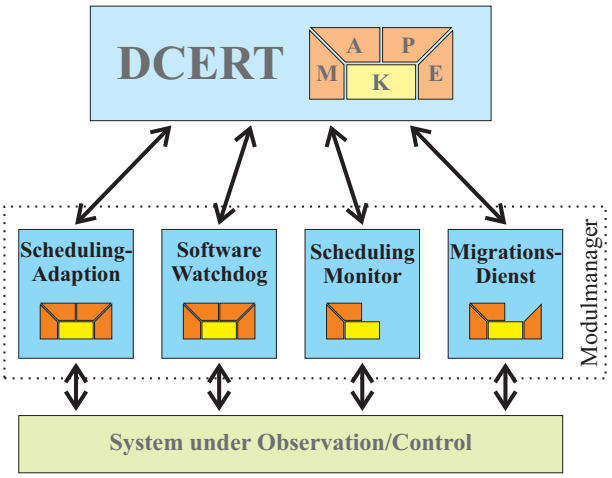


Abbildung 6.1: Basisszenario zur Evaluierung von DCERT

In Abschnitt 6.1 wird ein Modulmanager zur Schedulingadaption vorgestellt, mit dessen Hilfe der Energieverbrauch eines eingebetteten Systems optimiert werden kann. Abschnitt 6.2 beschreibt einen Software Watchdog, der Ausfälle von Anwendungen erkennen und beheben kann. Diese beiden Modulmanager besitzen eine eigene Entscheidungslogik und können prinzipiell auch ohne DCERT ein-

gesetzt werden. In Abschnitt 6.3 wird ein Modulmanager zur Überwachung des CarCore-Hardware-Schedulers vorgestellt, der nur aus einem Monitor mit Analysekomponente besteht, aber selbst keine Aktionen anbietet. Zuletzt wird in Abschnitt 6.4 ein Migrationsdienst beschrieben, der Anwendungen zwischen Knoten eines verteilten Systems verschieben kann. Der Migrationsdienst stellt DCERT dazu Informationen über anstehende Migrationen sowie entsprechende Aktionen zur Verfügung. Er besitzt aber keine Logik, um über die Durchführung von Migrationen zu entscheiden, sondern ist hier auf DCERT und dessen genaueres Bild des Systemzustands angewiesen. Zu jedem Modulmanager werden kurz die Konzepte besprochen, nach denen er mit DCERT zusammenarbeitet. Die Formalisierungen hierzu finden sich in Kapitel 7 in der Vorstellung des Evaluierungsszenarios in Abschnitt 7.4.

6.1 Scheduling-Adaption zur Optimierung des Energieverbrauchs

Ein wichtiges Ziel beim Entwurf eingebetteter Systeme ist die effiziente Nutzung der zur Verfügung stehenden Rechenleistung. Der folgende Abschnitt beschäftigt sich mit der dynamischen Anpassung der Schedulingparameter für eine Klasse weicher Echtzeitanwendungen. Damit soll eine optimale Nutzung der verfügbaren Rechenleistung erreicht werden. In einem zweiten Schritt lässt sich damit der Energieverbrauch des Prozessors optimieren. Der hier vorgestellte Algorithmus eignet sich für einen Einsatz in Kombination mit DCERT, kann aber auch selbstständig betrieben werden [26, 25].

6.1.1 Einführung

Moderne Prozessoren bieten üblicherweise die Möglichkeit zur dynamischen Skalierung von Versorgungsspannung und Taktfrequenz (*Dynamic Voltage and Frequency Scaling, DVFS*). Zudem sind diese Prozessoren meist auch mit verschiedenen Schlafmodi ausgestattet, bei denen einzelne Teile des Prozessors komplett von der Versorgungsspannung getrennt werden.

Durch Herabsetzen der Taktfrequenz und gleichzeitiges Absenken der Versorgungsspannung lässt sich eine erhebliche Minderung der Leistungsaufnahme eines Prozessors erzielen. Tabelle 6.1 zeigt einen Auszug aus der elektrischen Spezifikation des Intel PXA270 (XScale). Ebenso sind dort die Leistungsdaten der verschiedenen Low-Power-Modi eingetragen.

Tabelle 6.1: Typische Werte der Leistungsaufnahme des Intel PXA270 (Auszug aus [18])

Parameter	P (mW)	$V_{\text{CC\_CORE}}$ (V)
624 MHz (208 MHz Systembus)	925,0	1,55
520 MHz (208 MHz Systembus)	747,0	1,45
416 MHz (208 MHz Systembus)	570,0	1,35
312 MHz (208 MHz Systembus)	390,0	1,25
208 MHz (208 MHz Systembus)	279,0	1,15
104 MHz (104 MHz Systembus)	116,0	0,9
13 MHz	44,2	0,85
Deep-Sleep	0,01014	0,0
Sleep	0,01630	0,0
Standby	1,7224	1,1

Da Prozessoren in den Schlafmodi normalerweise überproportional wenig Energie verbrauchen, werden in einfädigen Prozessoren Anwendungen nach dem *Race-to-Idle*-Verfahren ausgeführt, um Energie zu sparen. Solange der Prozessor mit der Abarbeitung der Anwendung beschäftigt ist, wird er bei maximaler Taktfrequenz betrieben. Damit wird der Anwendung eine maximale Rechenleistung zur Verfügung gestellt. Ist eine Teilaufgabe beendet und muss auf eine neue Aktivierung oder ähnliches gewartet werden, so wird der Prozessor in einen Schlafmodus versetzt. Damit wird gerade beim Auftreten längerer Rechenpausen weniger Energie verbraucht, als wenn der Prozessor dauerhaft bei einer niedrigeren Taktfrequenz betrieben würde.

In mehrfädigen Prozessoren mit mehreren aktiven (Echtzeit-)Anwendungen ist dies hingegen keine praktikable Vorgehensweise. Die laufenden Anwendungen können sich hinsichtlich ihrer Komplexität und ihrer Zeitschranken unterscheiden und laufen damit auch nicht synchron. Ein Schlafenlegen des Prozessors, nur weil eine Anwendung in der Ausführung pausieren muss, kann dazu führen, dass alle anderen Anwendungen ihre Zeitschranken verpassen. Stattdessen muss hier nach anderen Lösungswegen gesucht werden.

6.1.2 Problemdefinition

Die in diesem Abschnitt vorgestellte Methode befasst sich mit der Energieverbrauchsoptimierung für eine bestimmte Klasse weicher Echtzeitanwendungen. Nur diese bieten den Spielraum, der für diese Optimierungen nötig ist. Bei harten Echtzeitanforderungen gilt es, die Zeitschranke um jeden Preis einzuhalten. Ener-

gieverbrauch oder überschüssige Rechenleistung sind hierbei zweitrangig, stattdessen muss zwingend die Rechenleistung für den Fall des längsten Programmlaufs zur Verfügung stehen. Es muss also zwingend die Rechenleistung gemäß der WCET-Analyse bereitgestellt werden. Bei Anwendungen ohne Zeitschranken hingegen gibt es kein Gegengewicht zur Senkung der Rechenleistung. Damit würden diese prinzipiell mit minimaler Rechenleistung ausgeführt, was zu einer unerwünscht langen Laufzeit führen würde. Weiche Echtzeitanwendungen bieten die Möglichkeit, zwischen einer Minimierung der Rechenleistung auf der einen Seite und dem Einhalten der Zeitschranken auf der anderen Seite abzuwägen. Die überschüssige Rechenzeit steht dann zur Ausführung weiterer Anwendungen zur Verfügung. Falls es von der Hardware unterstützt wird, so kann aber auch die Taktfrequenz abgesenkt werden. Dies resultiert in Energieeinsparungen, was sich gerade in eingebetteten Systemen als Vorteil erweist.

6.1.2.1 Struktur der Anwendung

Eine typische Echtzeitanwendung mit weichen Zeitschranken ist die Dekodierung von komprimierten Audio- oder Videoströmen. Dabei handelt es sich um periodische iterative Prozesse. In jeder Iteration muss ein Datenrahmen dekodiert und dargestellt werden. Die Berechnungskomplexitäten der einzelnen Rahmen unterscheiden sich im Allgemeinen voneinander, folgen aber typischerweise einem periodischen Muster. Die Darstellung jedes Rahmens unterliegt dabei der gleichen Zeitschranke.

Algorithmus 6.1 Iterativer Prozess mit weicher Zeitschranke

```
loop
    iteration_type_1() ∨̇ iteration_type_2() ∨̇ ... ∨̇ iteration_type_n()
end loop
```

Algorithmus 6.1 zeigt die Grundstruktur des betrachteten Problems. Jeder Schleifendurchlauf besteht aus genau einer Iteration beliebiger Art (Typ 1 bis n). Die einzelnen Iterationstypen haben dabei unterschiedliche Laufzeiten, unterliegen aber derselben Zeitschranke. Auch die Laufzeiten desselben Iterationstyps unterliegen geringfügigen Schwankungen. Allerdings lassen sie sich klar zu anderen Typen abgrenzen. Diese Schwankungen sind die Folge unterschiedlicher Programmflüsse zur Laufzeit, etwa aufgrund unterschiedlicher Eingabedaten. Prinzipiell ist es auch möglich, dass mehrere Iterationstypen eine ähnliche Laufzeit haben. Algorithmus 6.2 zeigt einen beispielhaften Ablauf mit Periode 4. Der Iterationstyp 1 tritt dabei zusätzlich mit kleinerer Periode 2 auf.

Die Anwendung wird unter Einsatz einer echtzeitfähigen zeitbasierten Schedulingmethode ausgeführt. Damit ist eine objektive Messung der tatsächlichen

Algorithmus 6.2 Äußeres Erscheinen der weichen Echtzeitanwendung

```
⋮
iteration_type_1();
iteration_type_2();
iteration_type_1();
iteration_type_3();
iteration_type_1();
iteration_type_2();
iteration_type_1();
iteration_type_3();
⋮
```

Ausführungszeit sowie der ausgeführten Befehle möglich. Mithilfe eines Adaptionsmoduls werden zu Beginn jeder Iteration die Schedulingparameter so gesetzt, dass die Zeitschranke möglichst exakt eingehalten wird. Dazu soll nur eine minimale Instrumentierung der Anwendung vorgenommen werden, wie sie in Algorithmus 6.3 dargestellt ist. Durch die Instrumentierung wird dem Adaptionsmodul ausschließlich mitgeteilt, dass die vorherige Iteration beendet ist und nun eine neue beginnt. Ein weitergehender Informationsfluss findet nicht statt. Alle relevanten Informationen etwa über die Zeitschranke müssen bereits bei der Initialisierung des Adaptionsmoduls bereitgestellt werden. Die Anzahl der Iterationstypen sowie deren Periodizität sollen dagegen erst zur Laufzeit gelernt werden. Somit ist zur Nutzung des Adaptionsmoduls kein Detailwissen über die Anwendung nötig.

Algorithmus 6.3 Codestruktur

```
loop
    adaptation_instrumentation();
    iteration_type_1() ∨̇ iteration_type_2() ∨̇ ... ∨̇ iteration_type_n()
end loop
```

Durch diese lose Kopplung ergibt sich eine weitgehende Anwendungsunabhängigkeit des Adaptionsmoduls. Als einzige Einschränkung ist aber zu beachten, dass die Länge der einzelnen Iterationen beziehungsweise die durchschnittliche Iterationslänge in einem vertretbaren Verhältnis zum Rechenaufwand des Adaptionsmoduls stehen soll. Die Nutzung des Adaptionsmoduls setzt also eine gewisse Grundkomplexität der Anwendung voraus. Genauere Betrachtungen hierzu finden sich in der Evaluierung in Kapitel 7.

6.1.2.2 Lebenszyklus

Der Lebenszyklus des Adaptionsmoduls lässt sich in zwei sich abwechselnde Phasen unterteilen. In der *Lernphase* wird die Anwendung beobachtet und es werden Daten über deren Laufzeitverhalten gesammelt. Zum Ende der Lernphase werden aus diesen Daten die Verhaltensparameter des Adaptionsmoduls bestimmt. Die Aufgaben an dieser Stelle lassen sich grob wie folgt einteilen:

- Einteilung der Iterationen in Klassen mit ähnlicher Laufzeit und
- Erkennung von Regelmäßigkeiten und Periodizitäten in der Abfolge der Klassen.

In der nun folgenden *Arbeitsphase* werden diese Informationen genutzt, um Vorhersagen über das zukünftige Verhalten der Iterationen zu treffen. Parallel dazu werden weitere Informationen gesammelt, mit denen die Qualität der Verhaltensparameter eingeschätzt wird. Zu Beginn jeder Iteration werden die folgenden Schritte ausgeführt:

- Prüfen der Zeitschranke der vorhergehenden Iteration zur Qualitätsbestimmung, sowie
- Anpassung der Schedulingparameter für die folgende Iteration gemäß der Vorhersage.

Sollte die Vorhersagequalität unter eine vorgegebene Schranke sinken, so müssen entsprechende Maßnahmen ergriffen werden, um die Qualität wieder zu erhöhen. Ein erster Schritt hierzu ist ein Zurückfallen in die Lernphase, wodurch die Verhaltensparameter auf ein möglicherweise verändertes Verhalten der überwachten Anwendung angepasst werden. Eine weitergehende Maßnahme stellt die Anpassung der Lernparameter dar. So kann der Algorithmus etwa die Länge des Beobachtungsfensters erhöhen, um die folgenden Vorhersagen durch eine größere zugrundeliegende Datenbasis zu stabilisieren.

6.1.2.3 Bestimmung der Periodizität

Zur Bestimmung von Periodizitäten in Datenreihen werden hauptsächlich zwei eng miteinander verwandte Methoden verwendet. Zum einen ist dies die Berechnung von Periodogrammen auf Basis der *Diskreten Fourier-Transformation (DFT)*. Mit Hilfe des Periodogramms lassen sich dominante Frequenzen in der Datenreihe erkennen, woraus die zugehörige Periode bestimmt werden kann.

Bei der zweiten Methode handelt es sich um die *zyklische Autokorrelationsfunktion* über der Datenreihe. Deren Maxima weisen direkt auf Periodizitäten in der zugrundeliegenden Datenreihe hin. Die direkte Berechnung der Autokorrelation

über der Datenreihe $v_1, \ldots, v_n$ erfolgt nach folgender Formel:

$$AC(\tau) = \sum_{k=0}^{n-1} v_k v_{(k+\tau) \bmod n}, \quad \tau \in \{0 \ldots n-1\} \tag{6.1}$$

Eine Periodizität $P = \tau_{\max}$ ist bestimmt durch

$$AC(\tau_{\max}) = \max_{\tau \in \{1 \ldots n-1\}} AC(\tau) \tag{6.2}$$

Der Aufwand zur Bestimmung der Periodizität mittels der Autokorrelation liegt in $O(n^2)$. Daneben ist auch eine Berechung über die diskrete Fourier-Transformation der Datenreihe möglich. Kann dabei die schnelle Fourier-Transformation (*Fast Fourier-Transformation, FFT*) eingesetzt werden, so liegt der Berechnungsaufwand lediglich in $O(n \log n)$. Dies ist allerdings nur für Datenreihen mit Längen 2^n möglich. Eine solche Einschränkung ist aber nicht mit der hier geforderten Flexibilität vereinbar, weshalb beim Entwurf des Algorithmus zur Scheduling-Adaption die zyklische Autokorrelation zum Einsatz kommen wird. Eine detaillierte Betrachtung der hier genannten Techniken findet sich in Anhang A.

Für alle Techniken zur Bestimmung einer Periodizität ergibt sich aus dem Nyquist-Shannon-Abtasttheorem [42, 57] eine Einschränkung. Das Abtasttheorem besagt, dass bei einer Abtastfrequenz f maximal eine Frequenz von $f/2$ erkannt werden kann. Auf den vorliegenden Fall übertragen bedeutet dies, dass beim Vorliegen von n Werten eine Periode von höchstens $n/2$ erkannt werden kann.

6.1.3 Algorithmus: Autocorrelation Clustering

Auf Basis der zyklischen Autokorrelation führt dieser Abschnitt eine Adaptionsinstrumentierung gemäß Algorithmus 6.3 für periodische iterative weiche Echtzeitanwendungen ein. Namensgeber und Kern der Instrumentierung ist die zyklische Autokorrelation. Die Instrumentierung nutzt die über die zyklische Autokorrelation bestimmte Periode P, um P Cluster festzulegen. Die Elemente eines Cluster haben also alle die gleiche Position innerhalb der Perioden, und ähneln sich somit auch in ihrem Berechnungsaufwand.

Folgende Voraussetzungen müssen für den erfolgreichen Einsatz der hier beschriebenen Technik erfüllt sein:

- **Nachvollziehbarkeit** des Ausführungsverhaltens: am Ende einer Iteration muss festgestellt werden können, wie lange die Ausführung gedauert hat,

und welche Rechenleistung (z.B. Ausführungstakte) der Anwendung zur Verfügung standen.

- **Deterministisches zeitbasiertes Scheduling**: das Ausführungsverhalten der Anwendung muss sich auf Basis der oben erhobenen Daten vorhersagbar steuern lassen.

Das auf dem CarCore-Prozessor verfügbare *Periodic-Instruction-Quantum-Scheduling* (PIQ) erfüllt diese Voraussetzungen, da das *Instruction Quantum* eines Threads angibt, wie viele seiner Befehle in jeder Scheduling-Runde ausgeführt werden. Das PIQ-Scheduling dient im Folgenden als Basis für Algorithmus 6.4 zum *AutoCorrelation Clustering (ACC)*. Die Instrumentierung für das AutoCorrelation Clustering lässt sich in vier Schritte unterteilen:

1. **Aufzeichnung (Zeilen 2-5):** Die Laufzeit der vorhergehenden Iteration wird gemessen. Eine Echtzeituhr liefert hierfür eine von der Taktfrequenz unabhängige einheitliche Zeitbasis. Aus der Laufzeit und den aktuellen Scheduling-Parametern wird die Anzahl der ausgeführten Instruktionen angenähert. Ebenso wird kontrolliert, ob die Zeitschranke eingehalten wurde. Diese Information wird ebenfalls aufgezeichnet.

2. **Lernen: (Zeilen 6-10)** Wurden ausreichend Iterationen ausgeführt, so werden in diesem Schritt die Verhaltensparameter für die späteren Vorhersagen berechnet. Kern dieser Berechnung ist die Bestimmung der Periodizität über die zyklische Autokorrelationsfunktion. Außerdem wird die Anzahl der Instruktionen für jeden Iterationstyp auf Basis der aufgezeichneten Daten festgelegt.

3. **Warten (Zeilen 11-13):** Aufgrund von Schwankungen in der Laufzeit auch eines einzelnen Iterationstyps kann die Iteration schon vor Erreichen der Zeitschranke beendet sein. In diesem Fall wird der Prozess schlafengelegt. Die verbleibende Zeit bis zur Zeitschranke steht nun komplett anderen Anwendungen zur Verfügung.

4. **Vorhersage & Frequenzanpassung (Zeilen 14-19):** Wurden die Verhaltensparameter erfolgreich bestimmt, so erfolgt in diesem Schritt die Vorhersage des folgenden Iterationstyps. Unter Nutzung der Instruktionszahl dieses Typs sowie der Zeitschranke wird nun der Schedulingparameter so gesetzt, dass die Zeitschranke genau erreicht werden kann. Anschließend kann zusätzlich die Taktfreqenz so gesetzt werden, dass gerade nicht weniger als die benötigte Rechenzeit zur Verfügung steht.

Der Basisalgorithmus kann noch auf verschiedene Weise erweitert werden:

- **Sleep-ACC (S-ACC):** Aufgrund der impliziten Rundungen in den Zeilen 3 und 4 berechnet Algorithmus 6.4 den Scheduling-Parameter eher pes-

Algorithmus 6.4 Schedulingadaption mit ACC: Instrumentierung der Anwendung

```
1:  procedure ADAPTATION_INSTRUMENTATION
2:      rtc_now = RTC_CURR;                                  ▷ aktuelle Zeit
3:      diff = rtc_now - rtc_prev;                           ▷ bestimme Laufzeit
4:      n_instr = diff * my_IQ;                              ▷ ausgeführte Instruktionen
5:      instr_buff[n_iterations++] = n_instr;                ▷ Aufzeichnen
6:      if n_iterations > LEN then                           ▷ Beobachtungsfenster voll?
7:          period = autocorrelate(instr_buff);              ▷ bestimme Periodizität
8:          n_iteration = 0;                                 ▷ Zurücksetzen
9:          ac_done = true;
10:     end if
11:     if diff < DEADLINE then                              ▷ Freie Zeit bis Deadline?
12:         application_sleep(DEADLINE - diff);              ▷ warte bis zur nächsten
    Iteration
13:     end if
14:     if ac_done == true then                              ▷ Verhaltensparameter bestimmt?
15:         next_n_instr = instr_buff[(n_iterationr - period) % LEN];   ▷
    Vorhersage!
16:         my_IQ = next_n_instr / DEADLINE;  ▷ Berechne nötiges Instruction
    Quantum
17:         set_IQ(my_IQ);                                   ▷ Setze Instruction Quantum
18:         adjust_frequency(my_IQ);     ▷ Setze Taktfrequenz (nur bei einfädiger
    Programmausführung)
19:     end if
20:     rtc_prev = RTC_CURR;                                 ▷ ignore wait/sleep
21: end procedure
```

simistisch. Die Messung der Laufzeit basiert auf *Real-Time Clock Ticks*, von denen jeder aus mehreren tausend Taktzyklen besteht. Dadurch stellt der Algorithmus bei der Vorhersage (Zeilen 14-19) tendenziell zu viel Rechenzeit zur Verfügung, wodurch einzelne Iterationen möglicherweise deutlich vor ihrer Zeitschranke fertiggestellt werden. Die Erweiterung Sleep-ACC versetzt den Prozessor in dieser Zeit bis zum Beginn der nächsten Iteration in den Schlafmodus. Dazu muss der Aufruf in Zeile 12 durch `processor_sleep` ersetzt werden.

- **Race-ACC (R-ACC):** Falls keine andere Anwendung parallel läuft, kann die optimierte Anwendung die gesamte zur Verfügung stehende Rechenzeit ausnutzen. R-ACC arbeitet ähnlich wie S-ACC und passt die Taktfrequenz auf das notwendige Minimum an. Zusätzlich aber setzt es das Instruction Quantum so, dass die Applikation möglichst die gesamte Rechenzeit des

Prozessors ausnutzt. Algorithmus 6.5 zeigt, welche Änderungen dazu an dem Basisalgorithmus 6.4 vorgenommen werden müssen. Damit erreicht R-ACC ein Verhalten ähnlich zu Race-to-Idle, allerdings arbeitet der Prozessor hier üblicherweise bei einer niedrigeren Taktfrequenz.

- **Automatisches Neu-Lernen:** Falls Zeitschranken zu oft oder zu extrem verpasst werden, kann der Algorithmus eine erneute Lernphase einleiten. Dabei werden alle bisher erlernten Ergebnisse verworfen. Dies erweist sich dann als sinnvoll, wenn sich die Periodizität oder der Aufbau der Periode ändert, etwa durch Variationen in einem verarbeiteten Datenstrom.
- **Untere Schranke für Instruction Quantum:** Bei Iterationen mit sehr geringem Rechenaufwand kann es passieren, dass Algorithmus 6.4 in Zeile 16 ein sehr geringes Instruction Quantum errechnet. Da allerdings auch die Instrumentierung mit diesem Instruction Quantum ausgeführt wird, muss sichergestellt werden, dass dadurch keine Überschreitungen der Zeitschranken entstehen. Insofern empfiehlt es sich, ein minimales Instruction Quantum festzulegen und zu verwenden. Algorithmus 6.6 zeigt, wie Algorithmus 6.4 dazu abzuändern ist.

Algorithmus 6.5 Veränderungen an Algorithmus 6.4 für R-ACC, Ersetzen der Zeilen 17 und 18

```
17: adjust_frequency(my_IQ);                    ▷ setze Taktfrequenz
18: set_IQ(max_IQ_for_frequency);
```

Algorithmus 6.6 Nutzung eines minimalen Instruction Quantums

```
17: set_IQ(max(my_IQ, MIN_IQ));
```

6.1.4 Integration in DCERT

Wie bereits erwähnt, kann ACC auch alleinstehend betrieben werden und selbstständig Frequenzanpassungen vornehmen. In einem komplexen eingebetteten System werden aber auch andere Anwendungen laufen, die ebenfalls Rechenzeit benötigen. Hier würde eine selbstständige Anpassung der Taktfrequenz dann früher oder später zu Fehlfunktionen des Systems führen. Stattdessen wird eine weitere Entscheidungsebene benötigt, die die global benötigte Rechenleistung im Blick hat.

Die Lösung dieses Problems wird durch den Einsatz eines zeitbasierten Schedulingverfahrens erreicht. Da alle laufenden Anwendungen durch dieses Verfahren verwaltet werden, ist auch immer eine konkrete Information über die Belastung

des Prozessors vorhanden. Steigt diese Belastung beziehungsweise die angeforderte Rechenleistung über die aktuell tatsächlich vorhandene Leistung, so kann das globale Management benachrichtigt werden. Dieses kann dann in Abhängigkeit vom aktuellen gesamten Systemzustand und den zur Verfügung stehenden Aktoren entsprechende Maßnahmen einleiten. Darunter fällt etwa die Erhöhung des Prozessortakts, aber beispielsweise auch die Verlagerung von Anwendungen auf andere Knoten innerhalb eines Netzes. Entsprechend wird das globale Management auch benachrichtigt, wenn die Auslastung unter eine vorgegebene Schranke fällt, so dass ein Absenken der Taktfrequenz möglich wäre. Auch in diesem Fall ist eine Abwägung bezüglich des gesamten Systemzustands und der vorhandenen Aktoren möglich und nötig.

6.2 Software Watchdog zur Anwendungsüberwachung

Viele moderne Prozessoren besitzen einen sogenannten Watchdog. Dieser überwacht die Funktionsfähigkeit des Systems. Üblicherweise ist dazu in der Hardware ein Zähler implementiert, der im Zeitverlauf automatisch inkrementiert wird. Die Software muss diesen Zähler regelmäßig zurücksetzen. Bleibt dies aus, so löst der Watchdog bei Erreichen eines definierten Zählerstandes einen Neustart des Systems aus. Auf diese Weise kann eine einfache Fehlfunktion der Software behoben werden.

In einem mehrfädigen Prozessor bietet sich auch eine Implementierung eines Software-Watchdogs an. Dieser wird als Helper Thread in einem eigenen Thread-Slot ausgeführt und ist damit unabhängig von der eigentlichen Anwendung. Die Anwendung und der Watchdog besitzen gemeinsame Variablen *prev_heartbeat* und *curr_heartbeat*. Beim Erreichen von Kontrollpunkten schreibt die Anwendung den aktuellen und den vorherigen Zeitstempel in diese Variable gemäß Algorithmus 6.7. Die Implementierung dieses Algorithmus' muss dabei sicherstellen, dass die Zeilen 2 und 3 atomar ausgeführt werden, damit die im Speicher stehenden Werte immer konsistent sind. Hierzu können entweder Sperrvariablen oder geeignete Instruktionen genutzt werden.

Algorithmus 6.7 Heartbeat-Instrumentierung

```
1: procedure HEARTBEAT_TICK
2:     prev_heartbeat ← curr_heartbeat
3:     curr_heartbeat ← CURR_TIME
4: end procedure
```

Der Watchdog Service wird in einem separaten Thread ausgeführt. So wird er bei einer Fehlfunktion der Anwendung nicht beeinträchtigt und kann diese erkennen. Dazu liest der Watchdog Service die gemeinsamen Variablen in regelmäßigen Abständen und vergleicht deren Inhalt mit der vorgegebenen maximalen Ausführungszeit (Algorithmus 6.8). Ebenso vergleicht er den letzten Zeitstempel *curr_heartbeat* mit der aktuellen Systemzeit. Sollte einer dieser Vergleiche den Schluss erlauben, dass die Anwendung nicht mehr funktionsfähig ist, so wird diese neu gestartet. Der Watchdog Service zählt dabei mit, wie oft er die Anwendung schon neu gestartet hat. Falls dieser Zähler eine vorher festgelegte Schranke MAX_RESTARTS überschreitet, ruft er stattdessen eine vom Entwickler vorgegebene Funktion auf, die geeignetere Maßnahmen in die Wege leitet, um die Funktionsfähigkeit des Systems wiederherzustellen.

Algorithmus 6.8 Watchdog Service

Ensure: $restart\_ctr$ initialisiert, $MAX\_RESTARTS$ gesetzt

```
 1: procedure HEARTBEAT_CHECK(WCET)
 2:     ct ← CURR_TIME                          ▷ Lesen der aktuellen Zeit
 3:     chb ← curr_heartbeat                    ▷ ...und der Zeitstempel
 4:     phb ← prev_heartbeat
 5:     if chb − phb > WCET or chb + WCET > ct then
 6:         if restart_ctr < MAX_RESTARTS then
 7:             restart thread
 8:             restart_ctr = restart_ctr + 1
 9:         else
10:             perform application-specific action
11:         end if
12:     end if
13: end procedure
```

Bei der Implementierung muss sichergestellt werden, dass die Zeilen 2 bis 4 von Algorithmus 6.8 unmittelbar hintereinander ausgeführt werden. Nur so kann der Watchdog hier konsistente Werte erhalten. Wird der Watchdog zwischen diesen Zeilen länger unterbrochen, so kann es vorkommen, dass die Anwendung inzwischen einen neuen Heartbeat geschrieben hat. Der Watchdog hingegen arbeitet weiter mit teilweise alten Werten, aber wenigstens einem neuen Wert. Dadurch besteht die Gefahr, dass ein Neustart der Anwendung ausgelöst, obwohl diese noch voll funktionsfähig ist.

Durch die Virtualisierung der Rechenzeit auf einem mehrfädigen Prozessor ergeben sich somit auch neue Möglichkeiten für die Funktionsüberwachung von Anwendungen. Anstatt einen Neustart des gesamten Systems auszulösen, muss nur der betroffene Thread neu gestartet werden. Zusätzlich ist es hier aber auch möglich, eine sehr feine und dienstspezifische Reaktion vorzunehmen.

Der Watchdog-Service lässt sich leicht so erweitern, dass er nicht nur eine, sondern auch mehrere Anwendungen überwachen kann.

6.2.1 Integration mit DCERT

Die Angabe einer anwendungsspezifischen Reaktion auf Anwendungsausfälle stellt eine erste Verbindungsmöglichkeit zum Organic Management des Knotens dar. Anstatt einer direkten Reaktion kann hier auch eine Triggernachricht an den Knotenmanager gesendet werden. Dieser ist dann für eine weitergehende Reaktion zuständig. Der Watchdog-Dienst selbst kann aber auch um zusätzliche Zähler erweitert werden, die die bisherigen Ausfälle der beobachteten Anwendung aufsummieren. An dieser Stelle bietet sich eine weitere Anbindungsmöglichkeit an das Organic Management des Knotens. Sollten solche Ausfälle gehäuft auftreten, so kann auf Ebene des Watchdogs nicht mehr geeignet reagiert werden, stattdessen wird die Kontrolle an den mächtigeren Knotenmanager abgegeben.

6.3 Schedulingmonitor

Die Bestimmung der Prozessorlast allein anhand der dem Betriebssystem bekannten Schedulingparameter der laufenden Anwendungen ist mit gewissen Ungenauigkeiten verbunden. Dieser Abschnitt stellt deshalb eine genauere Methode zur Messung der Prozessorlast im Hinblick auf das PIQ-Scheduling vor.

Harte Echtzeitanwendungen werden mit Hilfe des DTS-Schedulings ausgeführt. Der Schedulingparameter gibt hier die Anzahl Takte je Schedulingrunde an, für die der Thread auf Hardwareebene maximale Priorität für all seine Operationen erhält. Aus den Schedulingparametern aller DTS-Threads kann somit errechnet werden, wieviel Rechenzeit je Runde *garantiert frei* ist. Tatsächlich aber steht aufgrund von Latenzen sogar noch mehr Zeit zur Verfügung, die aber von den ausgeführten Instruktionen der DTS-Threads abhängt. Diese freie Ausführungszeit füllt der Scheduler mit Threads nach dem PIQ- und RRIQ-Scheduling auf. Threads mit RRIQ-Scheduling (*Round Robin by Instruction Quantum*) haben keinerlei Zeitgarantien. Deshalb werden diese im Folgenden nicht weiter betrachtet.

Beim PIQ-Scheduling für weiche Echtzeitanwendungen gibt der Schedulingparameter vor, wie viele *Instruktionen* einer Anwendung je Zeiteinheit ausgeführt werden, eben das *Periodic Instruction Quantum*. Der Scheduler versucht, diese Anforderung zu erfüllen, kann aber auch keine absolute Garantie dafür geben. Dies begründet sich insbesondere darin, dass nicht jede Instruktion die gleiche Ausführungszeit beansprucht. In einem mehrfädigen superskalaren Prozessor

können außerdem mehrere Instruktionen gleichzeitig ausgeführt werden. Ob und in welcher Konstellation diese Ausführung stattfindet, hängt auch von den anderen parallel laufenden Anwendungen und deren Priorität ab.

Der Schedulingmonitor besteht aus zwei Modulen: eine Hardware-Erweiterung, die das Ausführungsverhalten aller PIQ-Threads überwacht, sowie dem Software-Modulmanager, der eine feinere Auswertung der von der Hardware gesammelten Informationen vornimmt.

6.3.1 Hardware-Erweiterung

Die Hardware-Erweiterung ist mit dem Scheduler des CarCore verbunden. Der Hardware-Scheduler liefert am Ende einer jeden Scheduling-Runde zwei Werte EARLY_SATURATION und UNSAT_QUANTUM (siehe Abschnitt 2.2.3.3).

Die Hardware-Komponente des Schedulingmonitors summiert die auflaufenden Werte dieser Register auf und hält sie, ebenso wie die Rundenzahl n, auf die sich diese Summen beziehen, für die Software bereit. Nach Lesezugriffen setzt sie alle diese Werte automatisch auf Null zurück.

6.3.2 Schedulingmonitor

Der zugehöriger Modulmanager besteht nur aus einer Monitorkomponente, dem eigentlichen Schedulingmonitor. Er wertet die vom Hardwaremodul aufgezeichneten Daten aus und sendet bei Bedarf entsprechende Nachrichten an DCERT. Dazu berechnet der Schedulingmonitor jeweils die Durchschnittswerte pro Runde von EARLY_SATURATION und UNSAT_QUANTUM und vergleicht diese mit einer vorgegebenen Schranke BOUND:

- SUM_EARLY_SATURATION / n > BOUND
  ⇒ Nachricht `HIGH_PERFORMANCE`
- SUM_UNSAT_QUANTUM / n > BOUND
  ⇒ Nachricht `LOW_PERFORMANCE`

Für die Wahl von BOUND ist es von Vorteil, wenn der Schedulingmonitor Kenntnis über das Frequenzskalierungsverhalten des Prozessors hat. Durch Veränderung der Rundenlänge des Schedulers proportional zur aktuellen Taktfrequenz hat eine Schedulingrunde immer die gleiche Zeitdauer. Abbildung 6.2 verdeutlicht dieses Vorgehen. Ein Kästchen entspricht dabei einem Prozessortakt. Die Dauer einer Runde beträgt dabei immer t, die Taktzahl dagegen variiert in Abhängigkeit von der Taktfrequenz. Die Schranke BOUND kann nun so gewählt werden, dass sie der Anzahl Takte entspricht, die durch ein Erhöhen oder Senken der Taktfre-

quenz um eine Stufe gewonnen beziehungsweise aufgegeben würden. Auf diese Weise kann der Schedulingmonitor nur dann eine Frequenzänderung auslösen, wenn diese erheblichen Einfluss auf das Zeitverhalten der Anwendungen oder den Energieverbrauch des Prozessors hat. Gleichzeitig muss der Schedulingmonitor oder DCERT aber auch den Zeitbedarf laufender harter Echtzeitanwendungen beachten. Deren gesamtes Cycle Quantum gibt eine untere Schranke für die Rundenlänge und damit auch die Taktfrequenz vor.

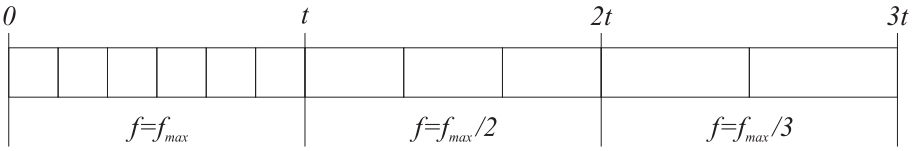


Abbildung 6.2: Prozessortakte und Rundenlänge bei verschiedenen Taktfrequenzen; jedes Kästchen steht für einen Prozessortakt

Gegenüber den hier betrachteten weichen Echtzeitanwendungen ist die großzügige Wahl von BOUND vertretbar. Insbesondere, wenn durch positives

$$\text{UNSAT\_QUANTUM} < \text{BOUND}$$

fehlende Rechenzeit angezeigt wird, ist doch ein gelegentliches und geringfügiges Verpassen von Zeitschranken zugunsten eines niedrigen Energieverbrauchs akzeptabel. Auch ein geringfügiger Überschuss an Rechenzeit, also

$$0 < \text{EARLY\_SATURATION} < \text{BOUND},$$

kann toleriert werden, da dieser auch von parallel laufenden Nicht-Echtzeit-Threads genutzt werden kann.

Positive Werte in den Monitorvariablen entstehen aufgrund einer nicht stetigen Programmstruktur zwangsläufig, ohne dass diese starke Auswirkungen auf das Ergebnis hätten. Oft sind deshalb nach jeder Schedulingrunde beide Variablen gesetzt, so dass zunächst gleichzeitig eine zu hohe und zu niedrige Performanz angezeigt wäre. Aber erst, wenn einer der beiden Zustände stark die Überhand gewinnt, also die Schranke BOUND übersteigt, geht der Schedulingmonitor von einem relevanten Einfluss auf das Gesamtverhalten aus. Wenn dieser Fall eintritt, so ist gleichzeitig die Wahrscheinlichkeit sehr gering, dass auch die jeweils andere Variable BOUND übersteigt.

6.4 Anwendungsmigration

Die CAROS-Architektur bietet durch den Runtime-Linker (Abschnitt 3.1.4) die Möglichkeit, zur Laufzeit eines Systems neue Applikationen auf einen Knoten

zu laden und auszuführen. Dieses Werkzeug kann auch dazu genutzt werden, um Anwendungen zwischen Knoten in einem verteilten System zu verschieben. Der folgende Abschnitt stellt einen *Migrationsdienst* vor, der diese Aufgabe in Zusammenarbeit mit DCERT übernimmt.

Die Möglichkeit, Anwendungen zwischen Knoten zu verschieben, erhöht die Flexibilität eines verteilten Systems. So ist es etwa möglich, durch gezieltes Verschieben von Anwendungen einzelne Knoten komplett von ihrer Rechenlast zu befreien. Diese können dann abgeschaltet werden, wodurch sich Energieeinsparungen erzielen lassen. Dies ist insbesondere in batteriebetriebenen Systemen von erheblichem Nutzen. Auf ähnliche Weise kann aber auch eine Lastverteilung in einem verteilten System erreicht werden, indem einzelne Anwendungen von stark belasteten Knoten auf solche mit geringer Auslastung verschoben werden. Zuletzt kann mit Hilfe des Migrationsdienstes auch die Funktionsfähigkeit eines verteilten Systems, in dem einzelne Knoten auszufallen drohen, aufrechterhalten werden. Hier werden Anwendungen von solchen kritischen Knoten auf andere, voll funktionsfähige Knoten verschoben.

Der Migrationsdienst basiert auf dem Runtime-Linker von CAROS (3.1.4 und 3.2.5). Er benutzt diesen zum Laden und Beenden von Anwendungen. Zusätzlich nutzt er eine Kommunikationsschnittstelle um Informationen sowie Anwendungscode und -daten mit den Migrationsdiensten anderer Knoten auszutauschen.

6.4.1 Migrationsprotokoll

Die Verschiebung einer Anwendung setzt eine vorhergehende erfolgreiche Verhandlung mit dem Empfängerknoten voraus. Nur wenn dieser über ausreichend freie Rechenkapazität verfügt, ergibt eine Migration überhaupt Sinn. Für die Verhandlungen mit möglichen Empfängern sowie die darauf folgende Migration verwendet der Migrationsdienst Nachrichten, deren Struktur in Abbildung 6.3 dargestellt ist. Jede Nachricht enthält zunächst die Adresse des Absenders und des Empfängers. Der Nachrichtentyp gibt an, welche Information mit dieser Nachricht bereitgestellt wird (siehe unten). Ein spezieller Migrationsvorgang wird durch eine eindeutig vergebene ID identifiziert, welche in allen Nachrichten einer Verhandlung identisch ist. Die Felder `src_idx` und `dst_idx` nutzen die Migrationmanager, um lokal eine schnelle Zuordnung der Nachrichten zu den zugehörigen Anwendungen vornehmen zu können. Zuletzt enthält dieser Nachrichtenkopf noch die Länge der gegebenenfalls angehängten Daten.

Die Migrationsverhandlungen finden auf Basis folgender Nachrichtentypen statt:

sender	receiver	type	id
timestamp	src_idx	dst_idx	data_length
data...			

Abbildung 6.3: Format der Migrationsnachrichten

MIGR_REQUEST Jede Migrationsverhandlung beginnt mit einer Migrationsanforderung, die der Migrationsdienst an alle ihm bekannten Knoten sendet. Diese Nachricht enthält im Datenteil zusätzliche Informationen über die Anwendung die verschoben werden soll, darunter etwa die von ihr erzeugte Rechenlast.

MIGR_PERMISSION Nach Empfang einer Migrationsanforderung kann ein Migrationsdienst, sofern alle Voraussetzungen erfüllt sind, die Verschiebung zu sich erlauben.

MIGR_DENIAL Falls ein Migrationsdienst nicht alle Voraussetzungen für eine Migration auf seinen Knoten erfüllt sieht, kann er die Migrationsanforderung stattdessen ablehnen.

MIGR_MIGRATION Empfängt ein Migrationsdienst eine Migrationserlaubnis, so hält er die Applikation an und versendet sie als Nachricht an den Empfängerknoten.

MIGR_CANCEL Unter Umständen erhält ein Migrationsdienst auf seine Migrationsanforderung hin mehrere positive Antworten, aber nur einer erhält tatsächlich die Anwendung. Alle anderen Knoten werden durch eine Abbruchnachricht davon in Kenntnis gesetzt, dass die Anwendung bereits erfolgreich versendet wurde.

Der Migrationsdienst stellt Schnittstellen bereit, mit denen zum einen eine Emigration initiiert werden kann, andererseits auch eine Immigration genehmigt werden kann. Der folgende Abschnitt stellt die Funktionsweise dieser Schnittstellen und des Migrationsprotokolls beispielhaft dar.

6.4.2 Integration mit DCERT

Zur Steuerung einer Migration stellt der Migrationsdienst zwei Aktoren bereit. Durch den Aufruf des *Emigrationsaktors* kann DCERT die Emigration, also das Verschieben einer Anwendung anstoßen, wenn etwa der Knoten überlastet ist. Für jede verschiebbare Anwendung sendet der Migrationsdienst auf Anfrage eine entsprechende Statusmeldung EMIG_STATE_x, die DCERT mitteilt, dass Anwendung x verschiebbar ist. Aufgrund dieser Nachrichten kann DCERT bei Be-

darf eine Anwendung auswählen, deren Migration durch den Emigrationsaktor gestartet wird.

Das Gegenstück hierzu ist der *Immigrationsaktor* sowie die Statusmeldung IMMIG_STATE_x, mit der der Migrationsdienst DCERT mitteilt, dass eine Migrationsanfrage vorliegt. DCERT trifft hier die Entscheidung, ob der Knoten in der Lage ist, diese Anfrage zu akzeptieren und ruft bei Bedarf den Immigrationsaktor auf, der dann alle weiteren Schritte veranlasst.

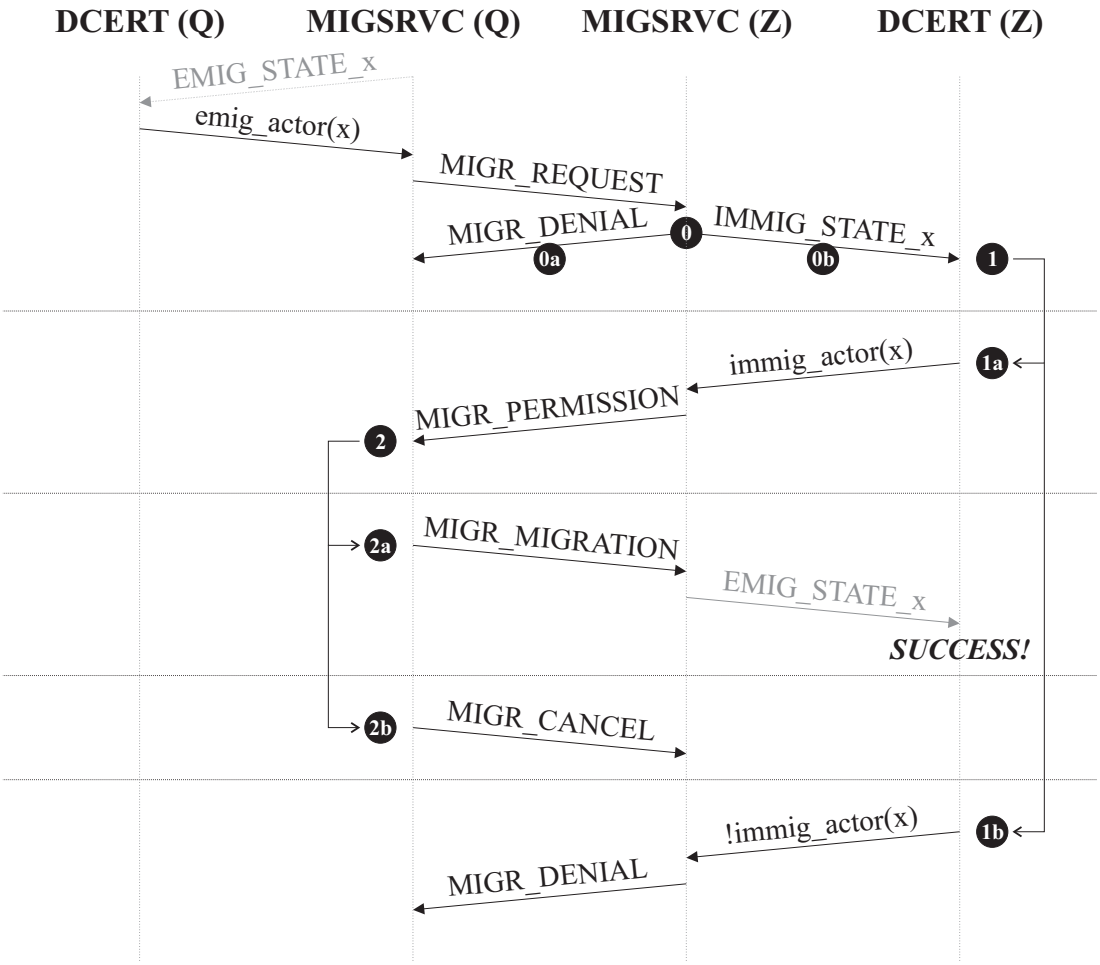


Abbildung 6.4: Ablauf der Anwendungsmigration

Der Ablauf einer gesamten Migration mit allen möglichen Ausgängen ist in Abbildung 6.4 als Sequenzdiagramm dargestellt. Voraussetzung für eine Migration ist immer, dass überhaupt eine migrierbare Anwendung vorhanden ist, also wenigstens ein Zustandsmeldung EMIG_STATE_x auf dem Quellknoten (Q) vom Migrationsdienst an DCERT gesendet wird. Falls es notwendig ist, ruft DCERT nun den Emigrationsaktor auf, welcher wieder eine Migrationsanforderung an alle

in Frage kommenden Knoten versendet. Die Migrationsdienste dieser potentiellen Zielknoten (Z) prüfen, ob sie prinzipiell in der Lage sind, diese Anwendung aufzunehmen. Fällt diese Prüfung negativ aus, so senden sie direkt eine Ablehnungsnachricht an den Quellknoten (Weg 0a). Ist die Prüfung hingegen erfolgreich (Weg 0b), so senden sie nun ihrerseits eine Statusmeldung IMMIG_STATE_x an das dortige DCERT. An dieser Stelle sind nun zwei Abläufe möglich.

DCERT kann die Anfrage ablehnen (Weg 1b), in diesem Fall wird eine Ablehnungsnachricht an den Quellknoten gesendet. Diese Ablehnung kann auch implizit erfolgen, indem DCERT nicht innerhalb einer vorgegebenen Zeit den Immigrationsaktor aufruft. Falls DCERT hingegen die Anfrage annimmt (Weg 1a), indem es den Immigrationsaktor aufruft, so sendet dieser eine Erlaubnisnachricht an den Quellknoten der Anfrage. Auch hier ergeben sich nun wieder zwei mögliche Abläufe.

Falls eine Erlaubnisnachricht die erste ist, die zu einem bestimmten Migrationsvorgang empfangen wurde, so hält der Migrationsdienst die ausgewählte Anwendung an und versendet sie an den Zielknoten. Der dortige Migrationsdienst bindet die Anwendung in den Knoten ein und startet sie wieder. Gleichzeitig setzt er nun auf seinem Knoten einen Zustand EMIG_STATE_x, der anzeigt, dass eine migrierbare Anwendung vorhanden ist. Die Migration ist dann erfolgreich abgeschlossen. Der Migrationsdienst des Quellknotens sendet nun noch Abbruchnachrichten an alle anderen Knoten, von denen er noch keine Antwort zu diesem Vorgang erhalten hat. Diese können dann unter Umständen für diesen Vorgang reservierte Ressourcen wieder freigeben.

100Beispielhafte Modulmanager für die Zusammenarbeit mit DCERT

7 Evaluierung

Das folgende Kapitel beschäftigt sich mit der Evaluierung der in den vorhergehenden Kapiteln beschriebenen Methoden. Der erste Teil dieses Kapitels befasst sich mit der Evaluierung jener Modulmanager aus Kapitel 6, die alleinstehend arbeiten können. Abschnitt 7.1 stellt die Anwendungsszenarien vor, die zur Evaluierung des Modulmanagers zur Scheduling-Adaption in Abschnitt 7.2 zum Einsatz kommen. In Abschnitt 7.3 wird der Software Watchdog zur Funktionsüberwachung von Anwendungen auf seine Eignung für Echtzeitsysteme untersucht. Der zweite Teil dieses Kapitels befasst sich mit der Evaluierung von DCERT und der Modulmanager, die auf DCERT angewiesen sind. Abschnitt 7.4 beschreibt die Formalisierung eines Autonomic Management auf Basis von DCERT und der vorgestellten Modulmanager. In Abschnitt 7.5 wird das Zusammenspiel von DCERT mit dem Modulmanager zur Schedulingadaption untersucht. Abschnitt 7.6 bewertet den auf DCERT angewiesenen Modulmanager zur Anwendungsmigration im Hinblick auf seinen Einsatz in Echtzeitsystemen. Abschnitt 7.7 schließt das Kapitel mit einer Betrachtung der durch DCERT entstehenden Kosten ab.

7.1 Szenarien

7.1.1 Matrixmultiplikation

Bei harten Echtzeitanwendungen muss sich die zur Verfügung gestellte Rechenzeit an der WCET der Anwendungen ausrichten. Dabei muss man davon ausgehen, dass diese Rechenzeit auch immer voll ausgenutzt wird. Da die wiederholte Multiplikation von Matrizen eine gleichmäßige Rechenlast erzeugt, eignet sie sich gut als Testanwendung. Aufgrund der einfachen Struktur des Multiplikationsalgorithmus' lässt sich dessen Laufzeit für beliebige Matrizengrößen sehr genau bestimmen. Durch Variation der Matrizengröße sowie durch Kombination verschiedener Matrizengrößen lässt sich eine Anwendung mit beliebiger Laufzeit und hoher Regelmäßigkeit erzeugen.

7.1.2 Video-Dekodierung (MPEG)

Die Dekodierung von Videos nach dem MPEG-2-Standard dient als Beispiel für eine weiche Echtzeitanwendung. Eine Videosequenz besteht aus mehreren Einzelbildern (*Frames*) aus den folgenden Klassen:

Intra Coded Pictures (I-Frames) sind Standbilder, die als Anker für wahlfreien Zugriff in den Videostrom dienen. Diese Frames haben nur eine geringe Kompression.

Predictive Coded Pictures (P-Frames) benötigen Informationen aus vorausgegangenen I- und P-Frames. Sie haben eine höhere Kompression im Vergleich zu den I-Frames.

Bidirectional Coded Pictures (B-Frames) benötigen Informationen aus vorausgegangenen und folgenden I- und P-Frames. Sie bieten die höchste Kompression.

Direct Coded Pictures (D-Frames) werden nur für den Schnellvorlauf benötigt. Es werden weniger Farbwerte gespeichert als bei den anderen drei Framearten. D-Frames werden im Folgenden nicht weiter beachtet, da sie nur in Ausnahmefällen Verwendung finden.

Die Übertragungsreihenfolge der Frames unterscheidet sich von deren Darstellungsreihenfolge. Da sich B-Frames auch auf folgende I- oder P-Frames beziehen, müssen diese schon weit vor ihrer Darstellung vorliegen. Aus diesem Grund ordnet die MPEG-Codierung den Datenstrom entsprechend um. Hat man etwa eine Darstellungsfolge $I_1 - B_1 - B_2 - P_1 - B_3 - B_4 - I_2$, so wird diese als $I_1 - P_1 - B_1 - B_2 - I_2 - B_3 - B_4$ übertragen.

Die Dekodierung der einzelnen Frames aus dem Datenstrom ist mit unterschiedlichem Rechenaufwand verbunden. Abbildung 7.1 zeigt beispielhaft die Berechnungszeiten für die einzelnen Frames einer Videosequenz. Für die Fertigstellung und Anzeige der Frames gilt immer die gleiche Zeitschranke. Bei einer üblichen Bildrate von 25 Frames pro Sekunde muss alle 40 ms ein neuer Frame angezeigt werden. Schwankungen in dieser Zeit können auftreten und werden toleriert. Allerdings leidet die Wiedergabequalität darunter.

Wie man in Abbildung 7.1 sieht, ist die Abfolge der verschiedenen Frame-Arten sehr regelmäßig. Ausnahmen gibt es nur zu Beginn einer Szene, wenn wie im obigen Beispiel für den ersten B-Frame zusätzlich noch ein P-Frame dekodiert werden muss. Hier kann die Darstellung etwas verspätet erfolgen. Im weiteren Verlauf fällt diese Umordnung aber nicht mehr ins Gewicht, da bereits berechnete Frames gepuffert werden. Die so eingesparte Berechnungszeit nutzt der Dekoder dann zur Berechnung späterer Frames.

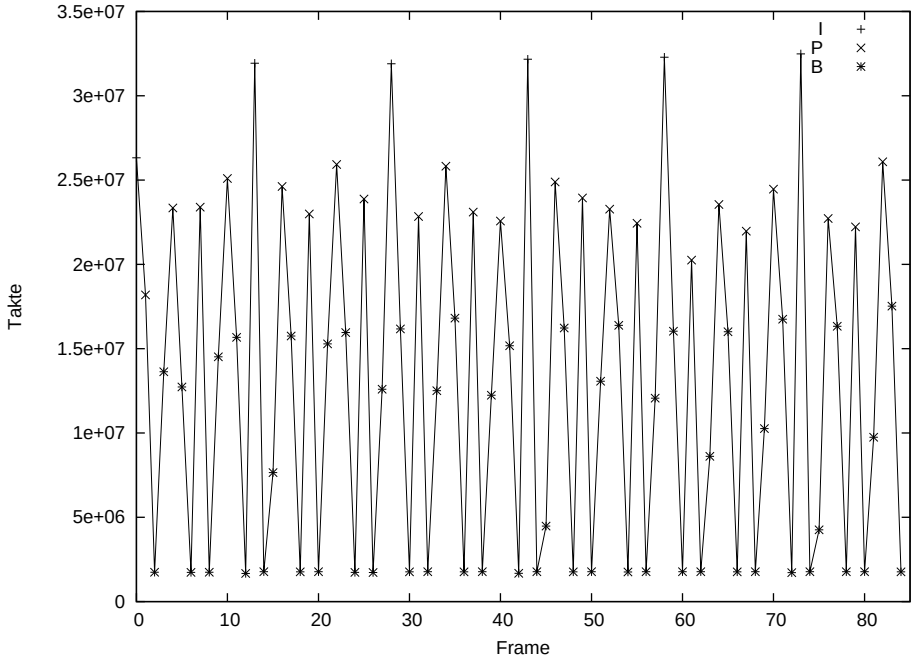


Abbildung 7.1: Laufzeiten bei der MPEG-Dekodierung

7.1.3 Energiemodell

Der CarCore-Prozessor verfügt über ein Modul, welches den Energieverbrauch des Prozessors bei unterschiedlichen Taktfrequenzen modelliert. Dieses Modul berechnet in jedem Takt die Leistungsaufnahme des Prozessors bei der gewählten Taktfrequenz und summiert diese Daten auf. Das Verhalten dieses Energiemoduls ist frei wählbar. Das für die folgenden Evaluierungen gewählte Frequenz- und Energiemodell orientiert sich am Verhalten des Intel XScale PXA270 [18]. Es verwendet die bereits in Tabelle 6.1 aufgeführten Kennwerte, implementiert aber nur einen Schlafmodus, hier *Deep-Sleep*.

7.2 Scheduling-Adaption zur Optimierung des Energieverbrauchs

Der Modulmanager zur Schedulingadaption, wie er in Abschnitt 6.1 vorgestellt wurde, kann sowohl alleinstehend als auch in Kombination mit DCERT eingesetzt werden. Im alleinstehenden Betrieb berechnet der Modulmanager zusätzlich zum Instruction Quantum der Anwendung noch die Taktfrequenz, die nötig ist, um dieses Instruction Quantum zu erfüllen. Hierzu nutzt er die in Abschnitt 6.3 vorgestellte Bindung der Scheduling-Rundenlänge an die Taktfrequenz. Auf dieser

Zeitbasis misst er die Ausführungszeit der einzelnen Iterationen und berechnet über das aktuelle Instruction Quantum die Anzahl der ausgeführten Instruktionen.

Als Maßstab für die folgenden Evaluierungen dient die verbreitete Scheduling-Technik *Race-to-Idle* (RTI). Da RTI nur bedingt für mehrfädige Echtzeitumgebungen geeignet ist, betrachten die folgenden Evaluierungen nur eine einfädige Programmausführung ohne parallel laufende Anwendungen. Die Evaluierung vergleicht RTI mit der Schedulingadaption mit Autocorrelation Clustering *ACC* sowie ihren Variationen *S-ACC* und *R-ACC*.

Die Länge *LEN* des Beobachtungsfensters muss abhängig von der erwarteten Periodenlänge gewählt werden. Auch während der Arbeitsphase zeichnet der Modulmanager fortlaufend Informationen über das Zeitverhalten der Anwendung auf.

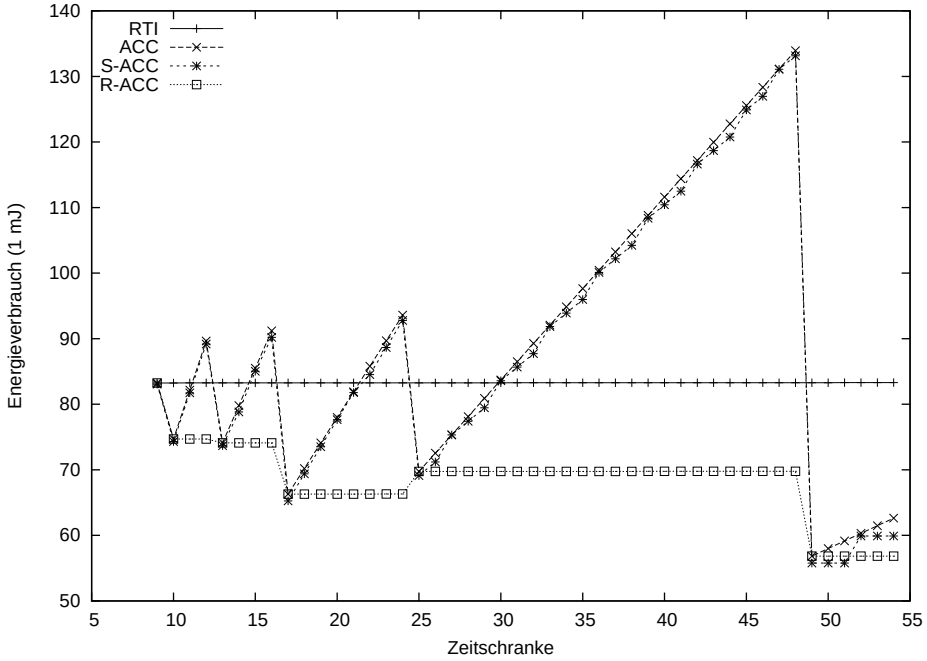
7.2.1 Abschätzung des Energieverbrauchs

Der erste Teil der Evaluierung befasst sich mit einer groben Abschätzung des Energieverbrauchs, der durch den Einsatz von RTI und der Variationen von ACC für ein gegebenes Programm resultiert. Diese Abschätzung wurde für die Ausführung einer festen Rechenlast unter variierenden Zeitschranken durchgeführt. Abbildung 7.2(a) zeigt das Ergebnis dieser Abschätzung.

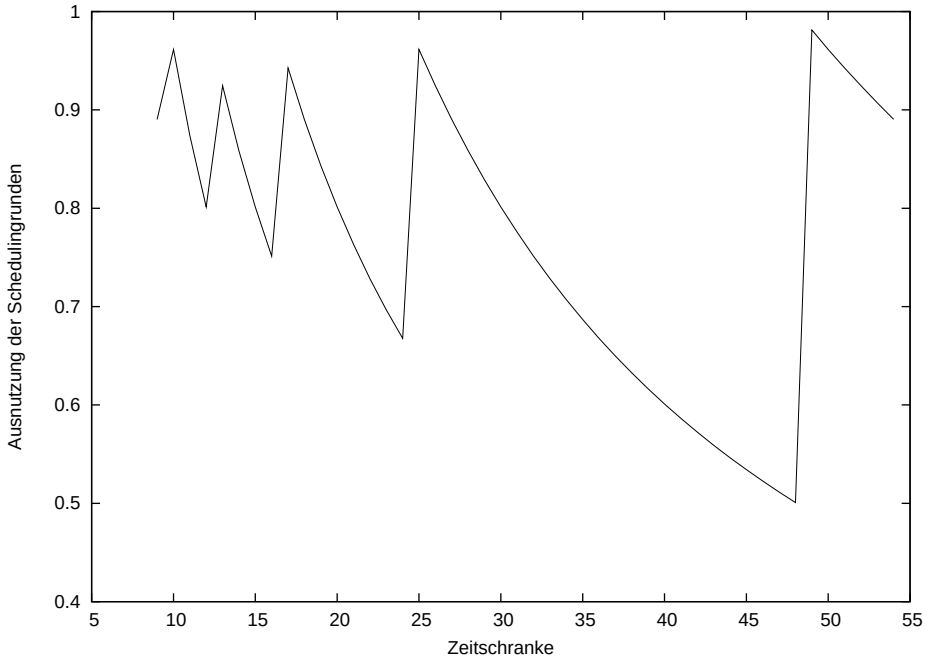
Der Energieverbrauch unter Einsatz von RTI bleibt unabhängig von der Zeitschranke nahezu konstant. Die Anwendung wird hier immer mit maximaler Leistung ausgeführt, was auch immer zu dem gleichen Energieverbrauch für die Berechnung führt. Die sich daran anschließenden Schlafphasen haben dagegen aufgrund ihrer Energieeffizienz nahezu keinen Einfluss auf den Gesamtverbrauch.

ACC und S-ACC zeigen ein sehr ähnliches Verhalten. Die Energieeinsparung von S-ACC im Vergleich zu ACC ist nach dieser Abschätzung nur marginal. Unter bestimmten Umständen können diese beiden Methoden einen geringeren Energieverbrauch erzielen als RTI. Dies ist immer dann der Fall, wenn das gewählte Instruction Quantum die Schedulingrunde möglichst gut ausnutzt. Abbildung 7.2(b) zeigt das Verhältnis von Instruction Quantum zu Rundenlänge. An den Stellen, an denen diese Kurve starke Sprünge macht, konnte der Manager die Taktfrequenz im Vergleich zur nächstniedrigeren Zeitschranke um einen Schritt senken. Anhand dieser Kurve sieht man, dass ACC/S-ACC dann besser sind, wenn die Ausnutzung der Rechenzeit über ca. 80% liegt.

R-ACC schneidet in dieser Abschätzung noch besser ab als RTI. Zwar sind die Arbeitsphasen bei R-ACC länger als bei RTI, aufgrund der nichtlinearen Abhängigkeit der Leistungsaufnahme von der Taktfrequenz verbraucht aber R-ACC für die



(a) Abschätzung des Energieverbrauchs



(b) Prozessorauslastung bei ACC/S-ACC

Abbildung 7.2: Abschätzung des Energieverbrauchs unter Einsatz von RTI und der ACC-Variationen

Tabelle 7.1: Belegungen für die Matrizenmultiplikation, in Klammern die einfädigen Laufzeiten in Takten

Typ Nr.	a		b		c		d	
I	7	(10432)	16	(117503)	19	(195237)	22	(301782)
II	8	(15455)	18	(166365)	22	(301782)	25	(440714)
III	9	(21635)	20	(227286)	25	(440714)	29	(685451)

gleiche Aufgabe weniger Energie. Da es die zur Verfügung stehende Rechenzeit optimal nutzt, kann R-ACC nach Berechnung des Ergebnisses noch eine Schlafphase bis zum Erreichen der Zeitschranke anschließen.

7.2.2 Untersuchungen an Matrixmultiplikationen

Ein zweiter Evaluierungsschritt hat das Ziel, ACC und die Scheduling-Adaption unter optimalen Bedingungen zu testen. In diesem Fall bedeutet dies die Anwendung auf ein iteratives periodisches Programm, bei dem die Iterationen einer Klasse exakt die gleiche Länge haben. Diese Forderung wird von dem Programm nach Algorithmus 7.1 erfüllt. Jede Matrixmultiplikation stellt dabei eine einzelne Iteration des Programms dar, die **for**-Schleife erzeugt die Periodizität. Das vorliegende Beispiel hat also eine Periodenlänge von 6.

Algorithmus 7.1 Matrizenmultiplikation

Ensure: matmul(n) multipliziert zwei $n \times n$-Matrizen miteinander; max_iterationen gesetzt

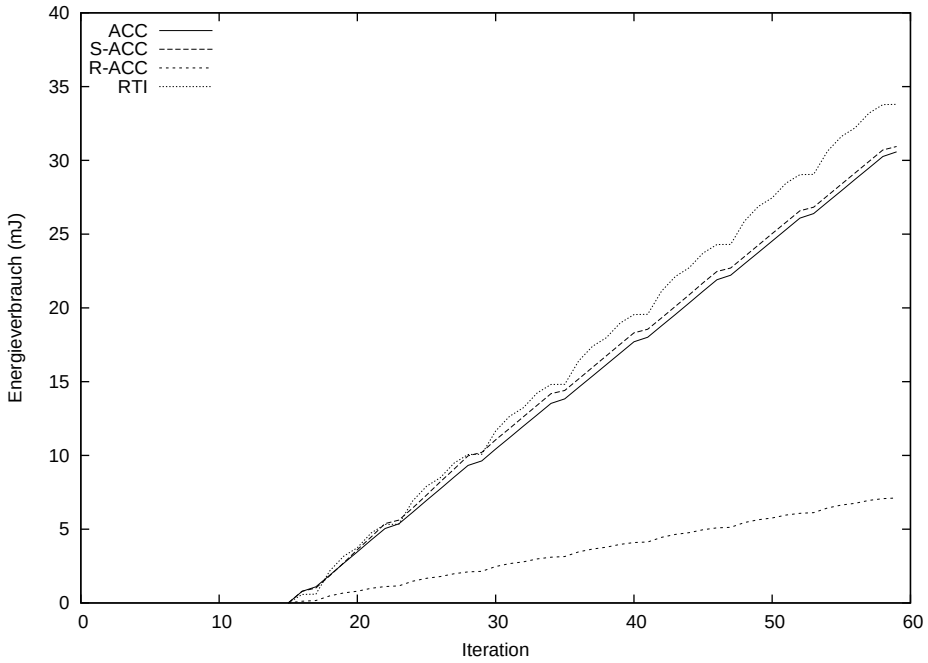
```
1: Adaption_init();
2: for i = 0 to max_iterationen do
3:     matmul(d); AdaptionsInstrumentierung();
4:     matmul(c); AdaptionsInstrumentierung();
5:     matmul(b); AdaptionsInstrumentierung();
6:     matmul(c); AdaptionsInstrumentierung();
7:     matmul(b); AdaptionsInstrumentierung();
8:     matmul(a); AdaptionsInstrumentierung();
9: end for
```

Die Untersuchungen zum Energieverbrauch und Zeitverhalten wurden an drei Iterationstypen mit unterschiedlicher Komplexität durchgeführt. Die Komplexität von Algorithmus 7.1 bestimmt sich aus den Parametern a, b, c, d. Tabelle 7.1 zeigt die drei Belegungen, die in den folgenden Evaluierungen Verwendung finden. Algorithmus 7.1 wurde mit diesen Belegungen und unterschiedlichen Zeitschranken ausgeführt, *max_iterationen* war dabei auf 10 begrenzt. Die Länge des Beobachtungsfensters beträgt hier 15 Iterationen. Die Periodenlänge von 6 wurde damit

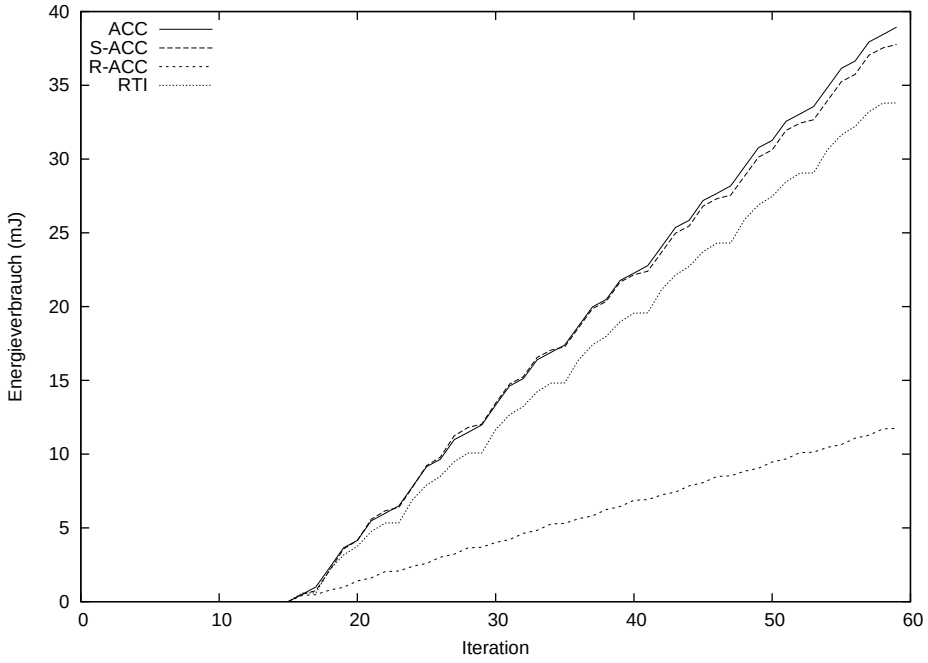
immer korrekt erkannt. Die Zeitschranken für die folgenden Evaluierungen sind immer in RTC-Ticks angegeben. Ein RTC-Tick ist immer gleichbedeutend mit einer Scheduling-Runde des Hardware-Schedulers und modelliert hier eine Zeitdauer von 100 μs, was in dem gewählten Prozessor- und Energiemodell 62400 Takten bei 624 MHz entspricht. Die in den folgenden Evaluierungen verwendete ACC-Instrumentierung implementiert in allen Variationen die Erweiterungen *Automatisches Neu-Lernen* und *Untere Schranke für Instruction Quantum* aus Abschnitt 6.1, Seite 88. Falls also die Zeitschranke zu oft oder zu stark verpasst wird, wechselt ACC zurück in die Lernphase. Die untere Schranke für das Instruction Quantum beträgt hier 100.

Bei jeder Belegung lässt sich ein Zeitschrankenbereich identifizieren, in dem ein Übergang von ACC/S-ACC zu RTI als effizientere Methode stattfindet. Die Abbildungen 7.3, 7.4 und 7.5 zeigen diese Übergänge. Der gezeigte Energieverbrauch ist auf den Beginn der Arbeitsphase normalisiert. Bei der jeweils niedrigeren Zeitschranke (Abbildungen 7.3(a), 7.4(a), 7.5(a)) erzielt ACC/S-ACC den geringeren Zeitverbrauch, bei der höheren (Abbildungen 7.3(b), 7.4(b), 7.5(b)) entsprechend RTI. R-ACC stellt auch hier immer die effizienteste Methode dar, um den Energieverbrauch zu senken. Bezüglich des Effizienzübergangs bestätigt diese Evaluierung damit die Abschätzung des vorhergehenden Abschnitts. Aufgrund der unregelmäßigen Struktur der Anwendung lässt sich hier aber nur noch ein einziger Übergang in der Energieeffizienz feststellen. Der gesamte Energieverbrauch entsteht als Summe von einzelnen Iterationen, die mit der von ACC gewählten Taktfrequenz ausgeführt werden. Jeder Iterationtyp erzielt damit eine andere Prozessorauslastung, so dass in manchen Fällen auch eine Ausführung mit RTI effizienter wäre. Wenn allerdings insgesamt die Auslastung durch ACC/S-ACC höher ist, hier also möglichst viele Iterationen mit hoher Prozessorauslastung ausgeführt werden können, so schneidet ACC auch beim Energieverbrauch besser ab. Abbildung 7.6 verdeutlicht diesen Sachverhalt am Beispiel des Komplexitätstyps I und der Methoden RTI und S-ACC. Zum einen zeigen die Graphiken den Energieaufwand für die einzelnen Iterationen nach der Lernphase. Zusätzlich ist die Differenz im Energieverbrauch zwischen RTI und SACC eingetragen. Negative Werte bedeuten hier, dass RTI die entsprechende Iteration mit einem geringen Energieaufwand ausgeführt hat, positive stehen für eine höhere Energieeffizienz durch S-ACC.

Die Abbildungen 7.7, 7.8 und 7.9 zeigen das Zeitverhalten der gewählten Beispiele. Für jede Iteration ist deren Dauer aufgetragen, zusätzlich ist die jeweilige Deadline eingezeichnet. Gut zu erkennen ist hier das allmähliche Einpendeln von ACC und S-ACC nach öfterer Berechnung der Verhaltensparameter. Während in der ersten Arbeitsphase ab Iteration 16 noch vermehrt Zeitschranken verpasst werden, erfolgt in späteren Phasen eine weitgehende Angleichung an die Zeitschranke. Bei den Ausreißern nach unten, wo also die Berechnung schon deutlich

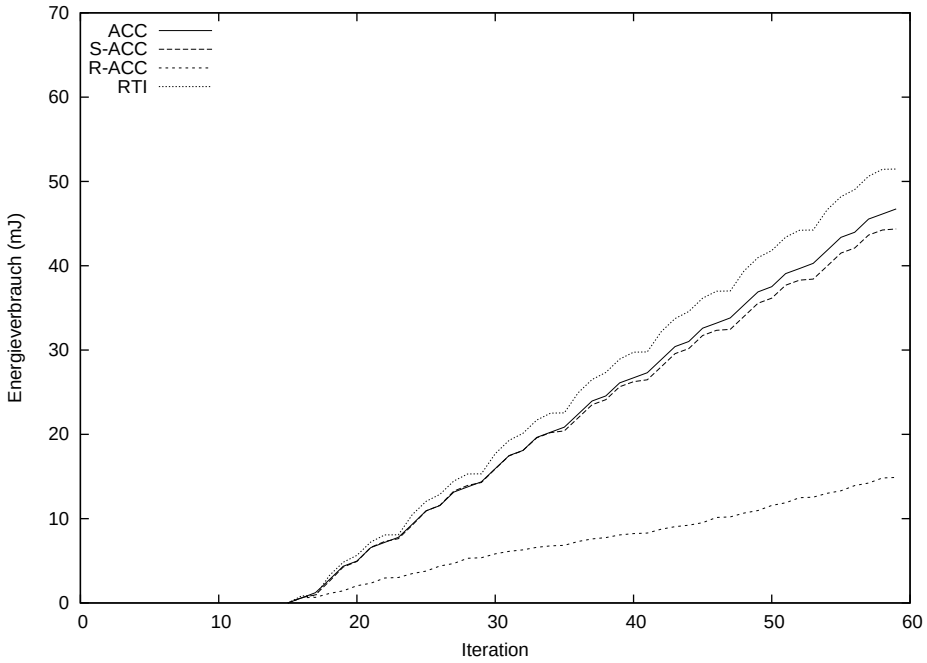


(a) Typ I, Deadline 75

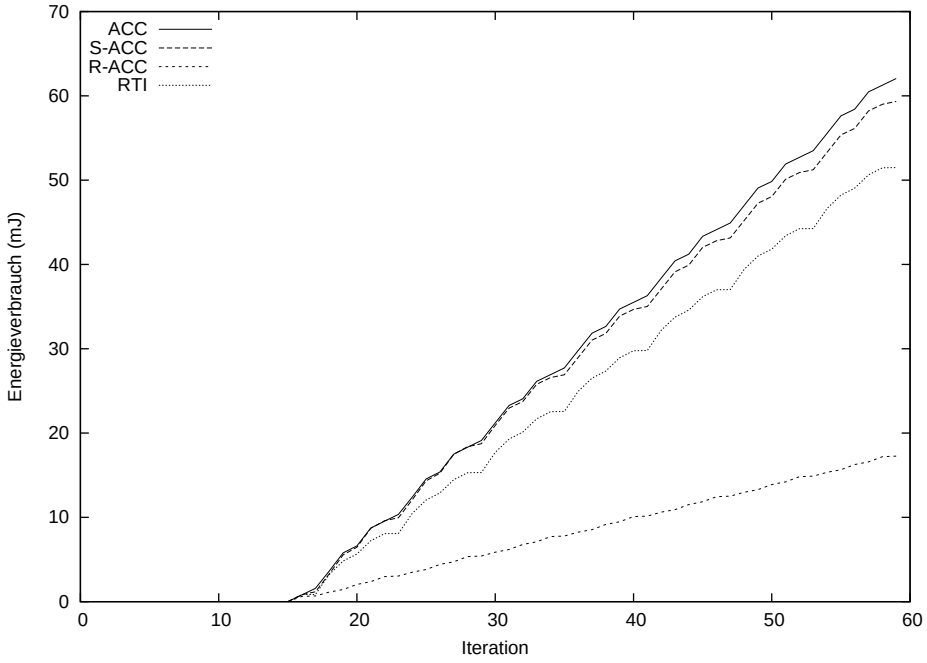


(b) Typ I, Deadline 125

Abbildung 7.3: Akkumulierter Energieverbrauch der Matrixmultiplikationen Typ I, gemessen ab dem Ende der Lernphase

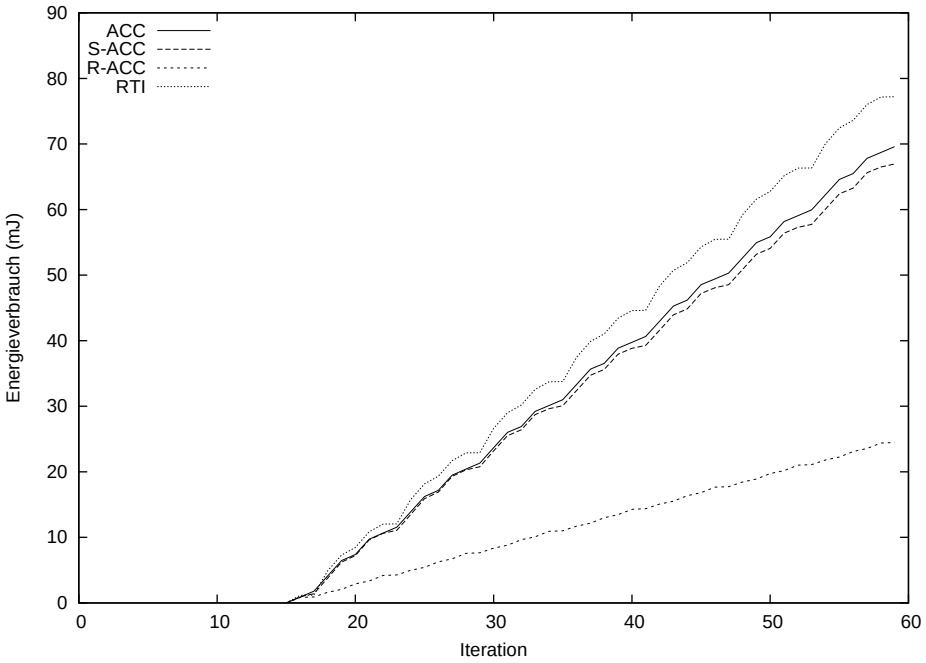


(a) Typ II, Deadline 150

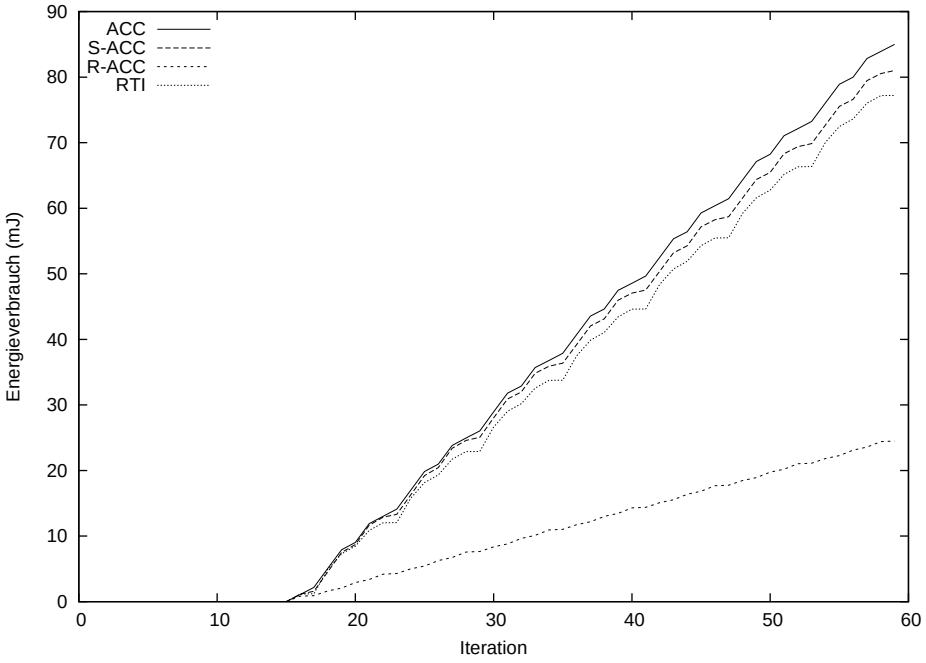


(b) Typ II, Deadline 200

Abbildung 7.4: Akkumulierter Energieverbrauch der Matrixmultiplikationen Typ II, gemessen ab dem Ende der Lernphase

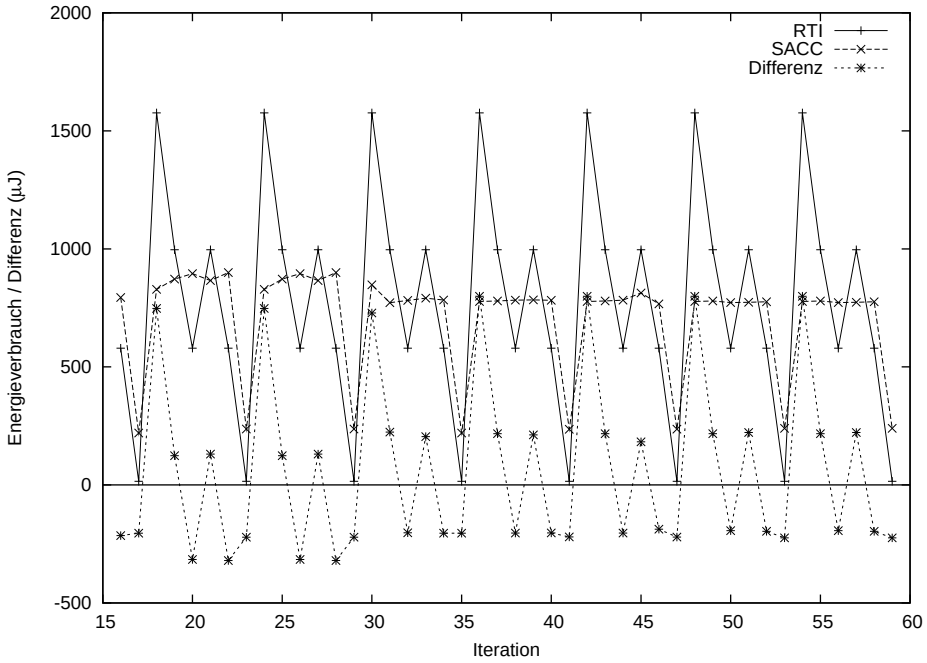


(a) Typ III, Deadline 225

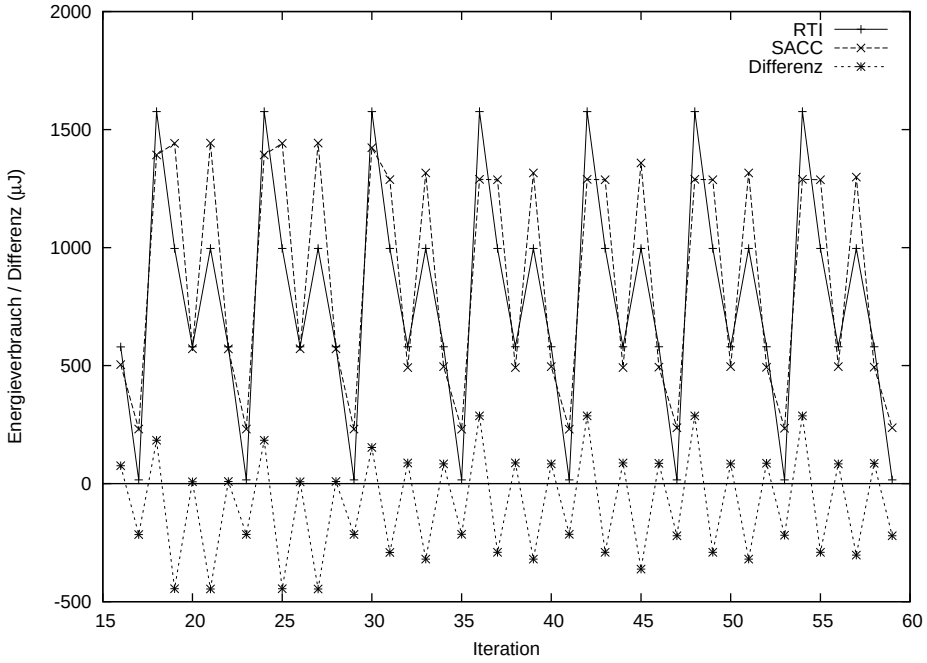


(b) Typ III, Deadline 275

Abbildung 7.5: Akkumulierter Energieverbrauch der Matrixmultiplikationen Typ III, gemessen ab dem Ende der Lernphase

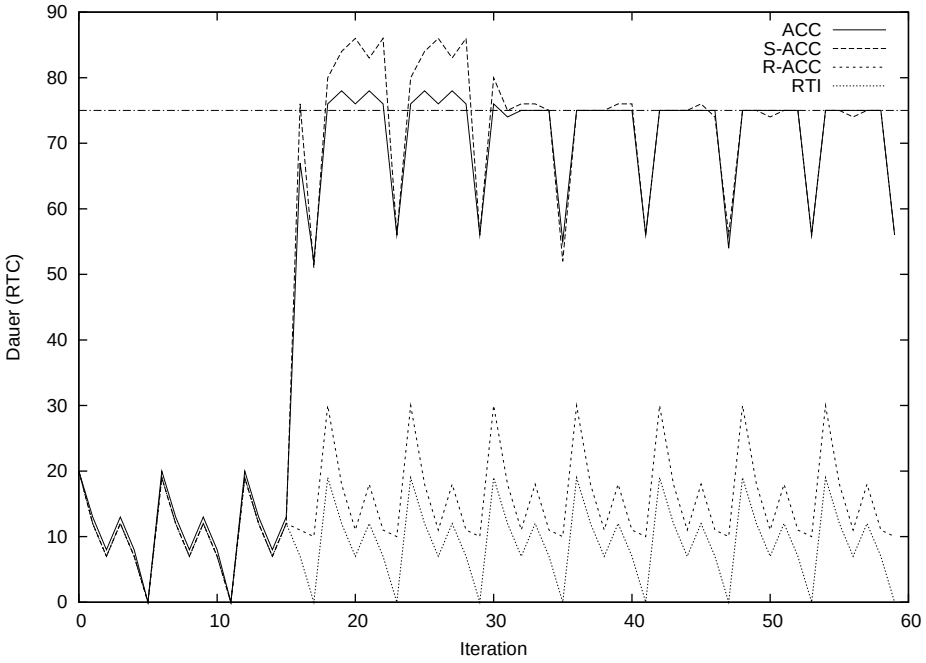


(a) Typ I, Deadline 75

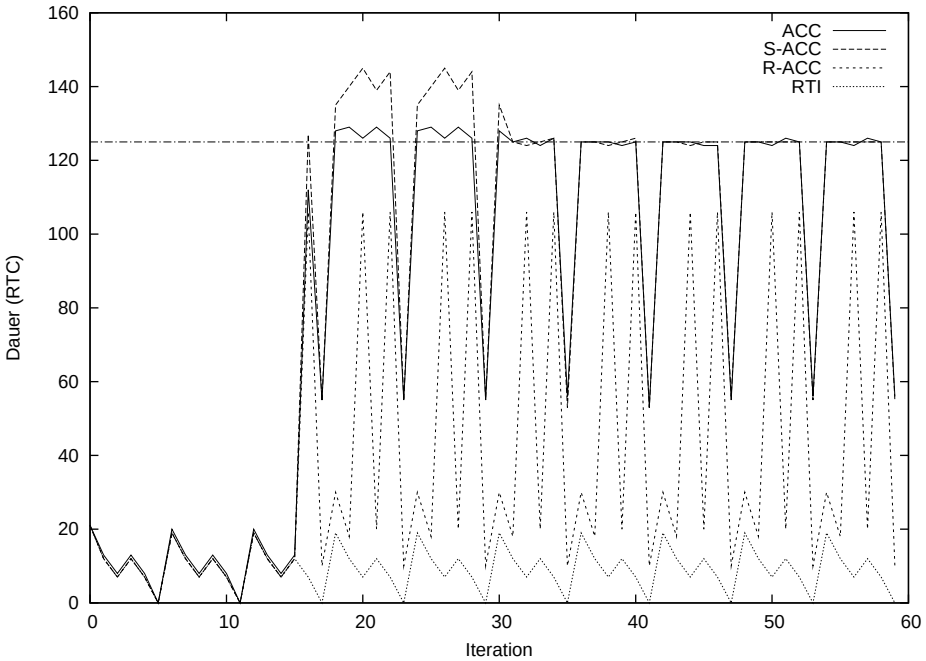


(b) Typ I, Deadline 125

Abbildung 7.6: Energieverbrauch einzelner Iterationen bei Optimierung mit RTI und SACC

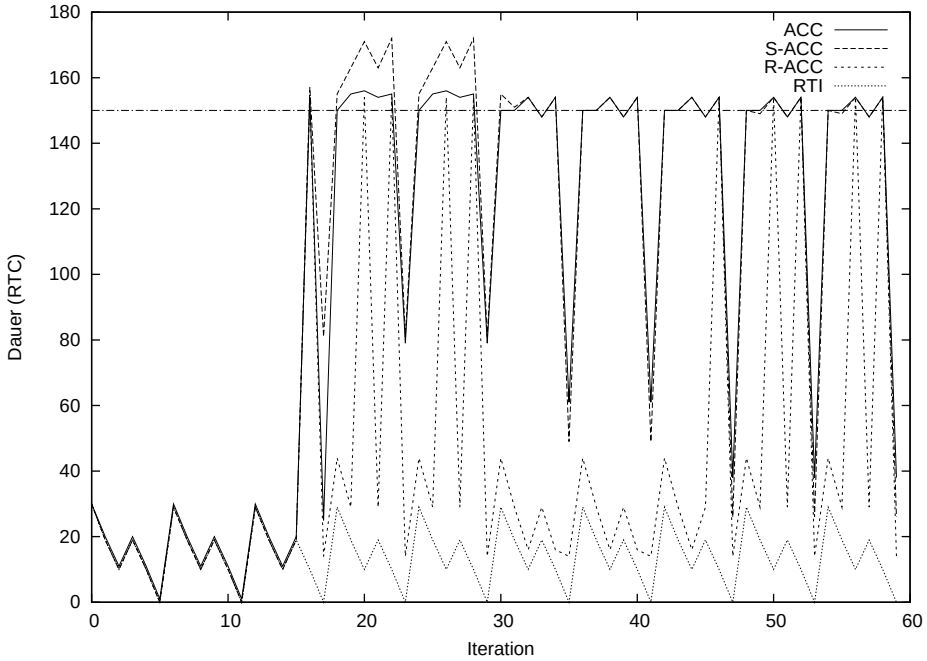


(a) Typ I, Deadline 075

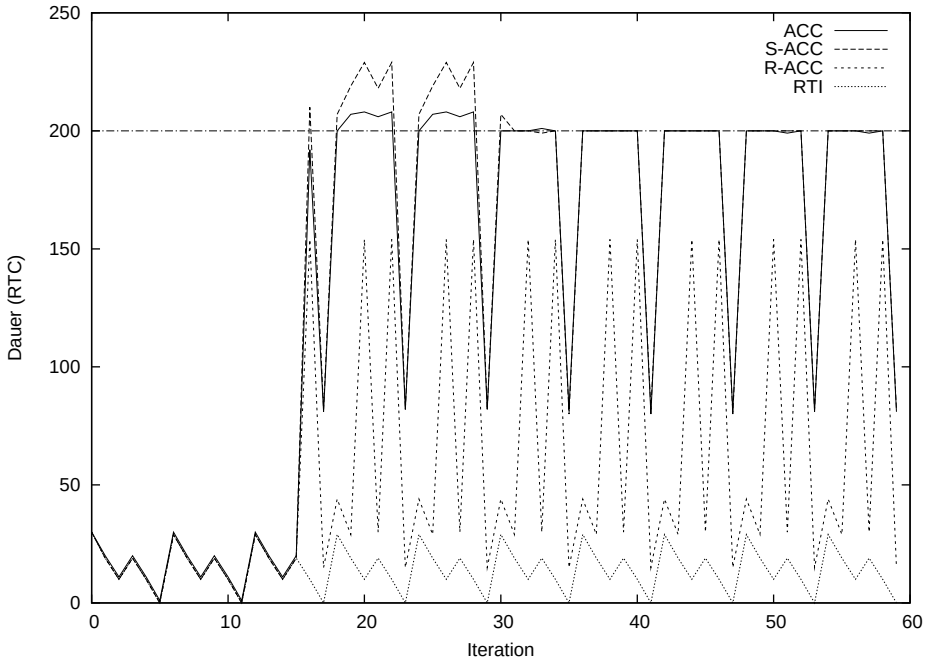


(b) Typ I, Deadline 125

Abbildung 7.7: Zeitverhalten der Matrixmultiplikationen Typ I

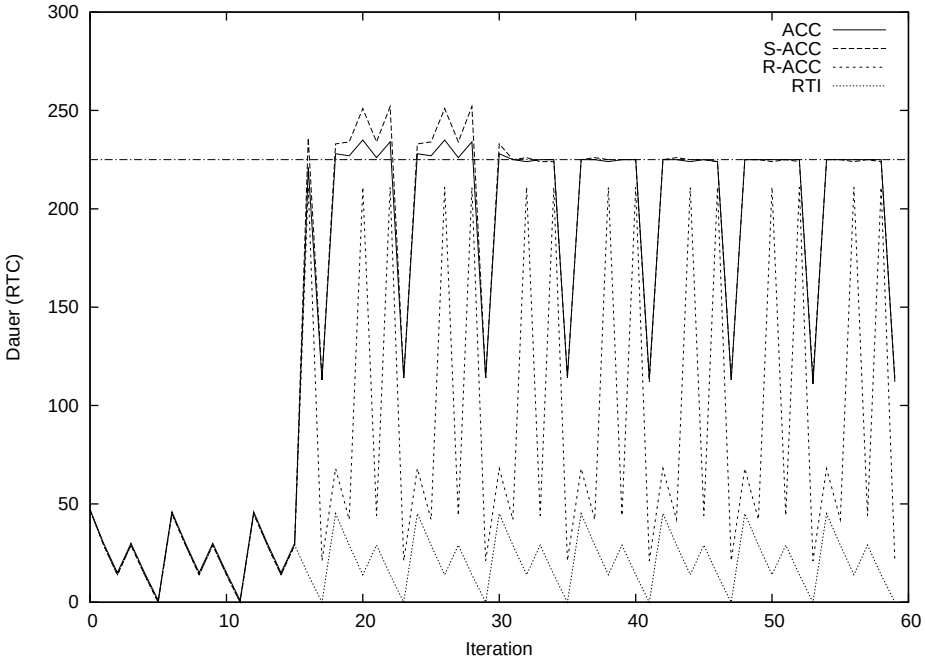


(a) Typ II, Deadline 150

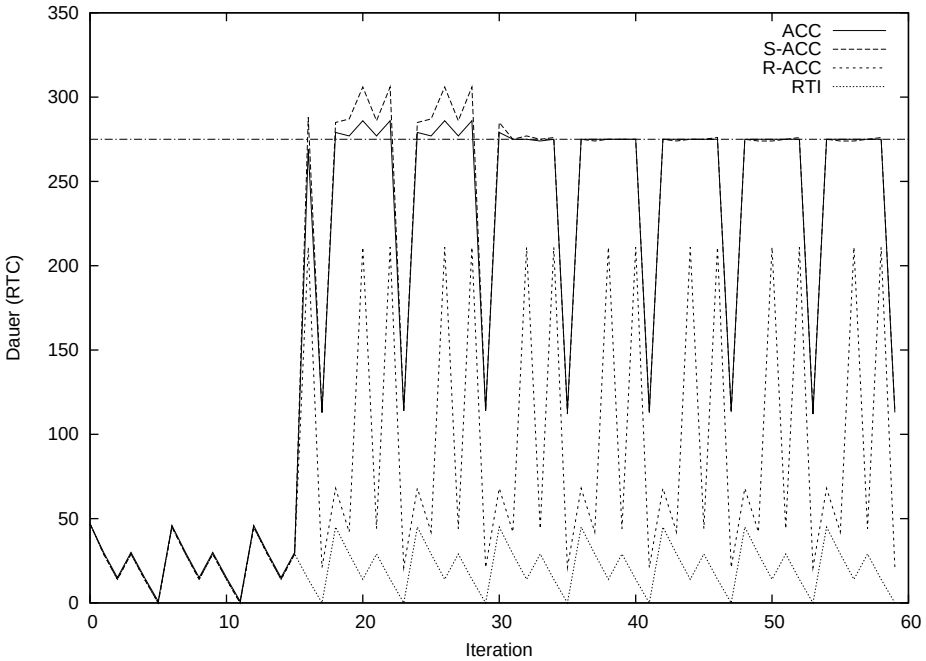


(b) Typ II, Deadline 200

Abbildung 7.8: Zeitverhalten der Matrixmultiplikationen Typ II



(a) Typ III, Deadline 225



(b) Typ III, Deadline 275

Abbildung 7.9: Zeitverhalten der Matrixmultiplikationen Typ III

vor der Zeitschranke beendet ist, handelt es sich um Iterationen mit sehr geringem Rechenaufwand. Hier kommen zwei Punkte zum Tragen. Zum einen kann die Taktfrequenz nur in diskreten Schritten variiert werden. Insbesondere gibt es eine minimale Taktfrequenz, die nicht unterschritten werden kann, auch wenn sie für einzelne Iterationen noch zu hoch wäre. Feinere Einstellungen der Rechenzeit eines Threads nimmt ACC anhand des Instruction Quantum des Threads vor. Aber auch hier lässt ACC keinen beliebig kleinen Wert zu, sondern garantiert ein minimales Instruction Quantum. Dadurch erhalten die wenig aufwändigen Iterationen mehr Rechenzeit, als sie eigentlich benötigen und sind entsprechend früh beendet.

Die Optimierung mit R-ACC zeigt im Fall von Abbildung 7.8(a) ein auffälliges Verhalten. Anstatt sich bei der Vorhersage auf eine bestimmte Wertefolge zu stabilisieren, entstehen hier durch die Rundungen in ACC starke Verhaltensschwankungen. Während der ersten Arbeitsphase verpasst R-ACC in einigen Iterationen die Zeitschranken knapp, hält sie hingegen in anderen sehr gut ein. In der zweiten Arbeitsphase hält es nun alle Zeitschranken ähnlich zu RTI sehr gut ein. Abbildung 7.10 zeigt das Verhalten dieser Konstellation für eine längere Laufzeit. Man sieht, das der Wechsel zwischen den beiden Arbeitsphasen anhält.

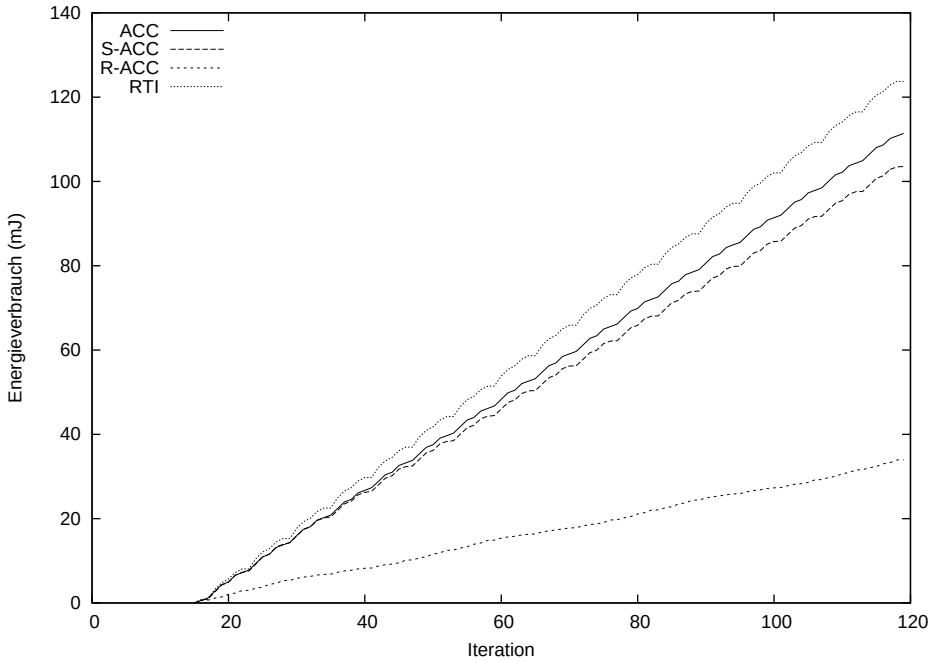
7.2.3 Praxistest: MPEG

Seine Praxistauglichkeit stellt die Scheduling-Adaption mit ACC an der Dekodierung von MPEG-Videos unter Beweis. Grundlage für die folgenden Evaluierungen sind kurze Videosequenzen nach dem MPEG-2-Standard. Tabelle 7.2 führt die im Folgenden relevanten Eigenschaften der eingesetzten Videosequenzen auf. Die minimale und maximale Laufzeit ist in RTC-Ticks angegeben und bezieht sich auf eine Ausführung des Dekoderprogramms mit maximaler Taktfrequenz und Rechenleistung.

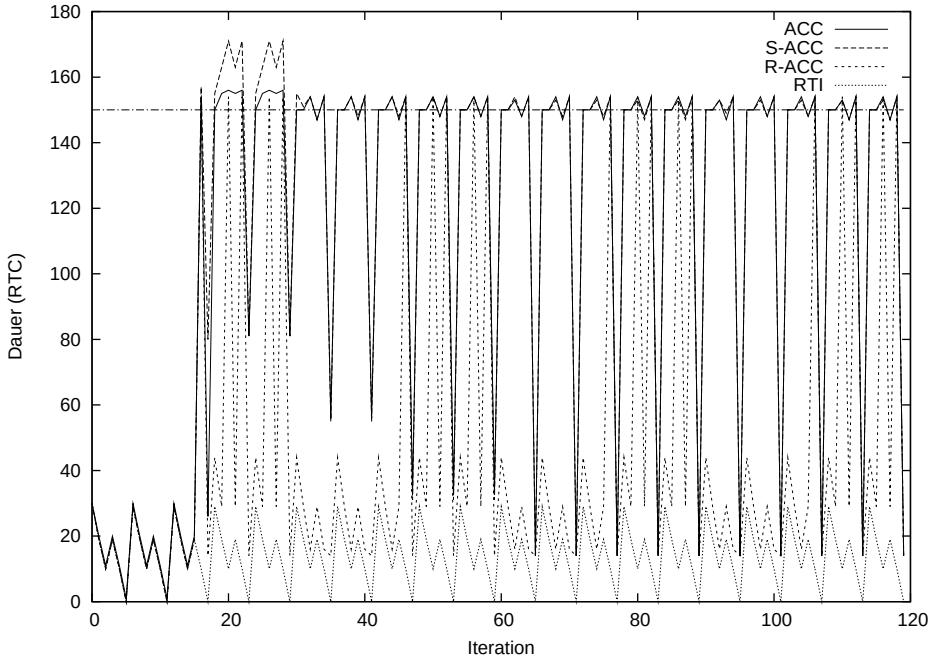
Tabelle 7.2: Eigenschaften der eingesetzten Videosequenzen

Videosequenz	SB	PS	WB
Anzahl Frames	85	142	153
min. Laufzeit (Ticks)	26	84	15
max. Laufzeit (Ticks)	442	365	451
Periodizität	15	12	18
erkannte Periode	6	6	18

Als Länge des Beobachtungsfensters wurde für diese Evaluierungen 40 gewählt. Gemäß dem Abtasttheorem können damit die vorliegenden Periodenlängen (siehe



(a) Energieverbrauch

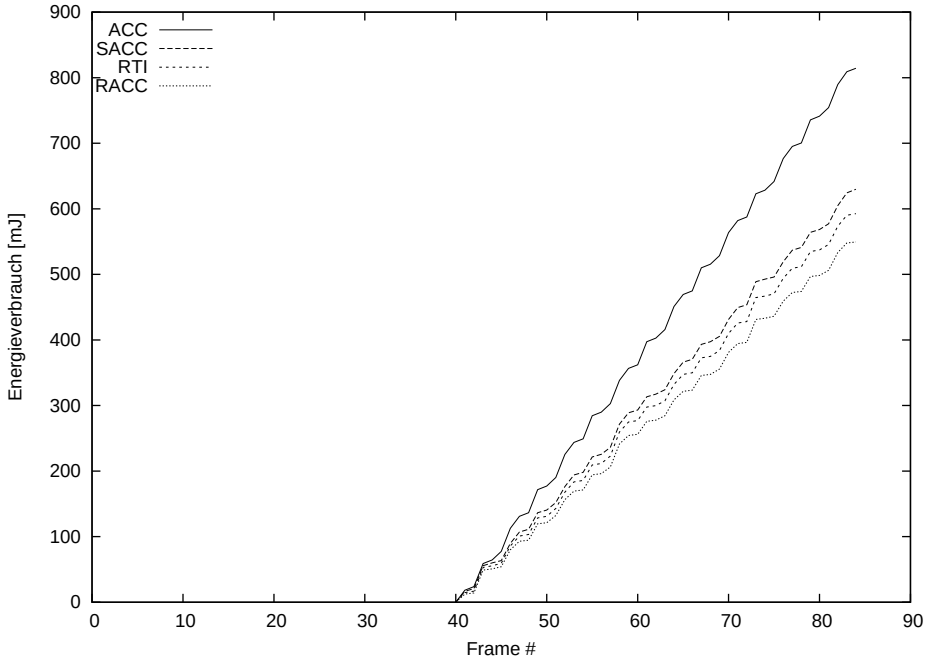


(b) Zeitverhalten

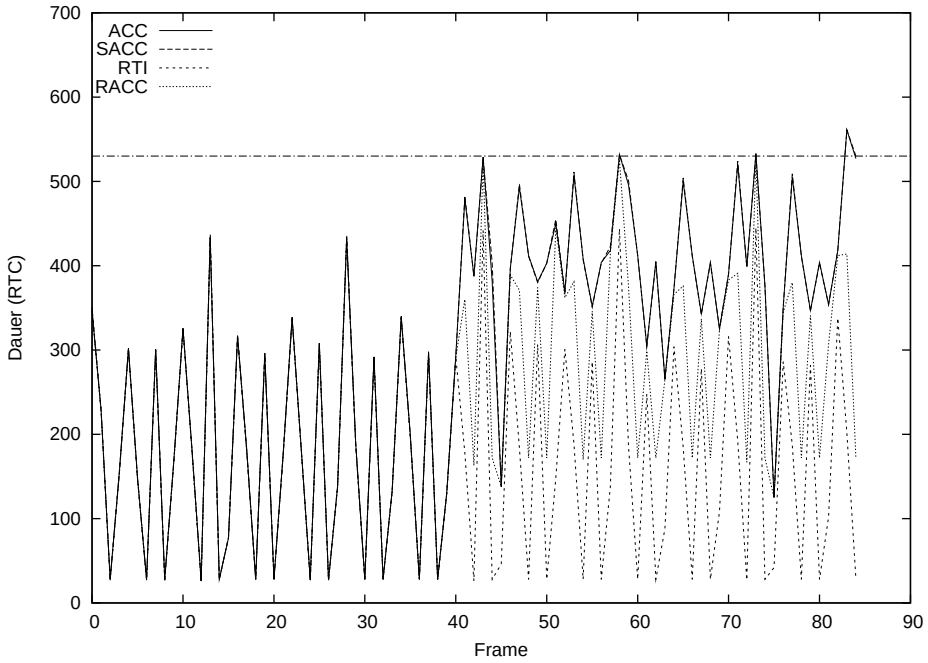
Abbildung 7.10: Matrixmultiplikation Typ II Deadline 150 mit längerer Laufzeit

Tabelle 7.2) erkannt werden. Die Periodenlänge ist hier der Abstand zwischen dem Auftreten zweier I-Frames. Tatsächlich aber erkennt ACC bei allen verwendeten Videosequenzen eine kürzere Periode von 6. Dies liegt daran, dass die I-Frames im Vergleich zu den P-Frames relativ selten auftreten und damit trotz ihrer höheren Laufzeiten in diesen untergehen. Ein ähnliches Verhalten wurde unter anderem auch schon von Vlachos et al. [64] beobachtet. ACC ordnet I- und P-Frames in gleiche Klassen ein, kann also diese beiden Frame-Typen nicht unterscheiden. Auf die Vorhersage hat dies insofern Auswirkungen, als dass ACC auch für P-Frames die Komplexität von I-Frames annimmt und damit häufig mehr als die benötigte Rechenleistung zur Verfügung stellt. Es hat sich aber gezeigt, dass auch die explizite Angabe der echten Periodenlänge anstatt der Berechnung durch die zyklische Autokorrelation keinen Einfluss auf die folgenden Ergebnisse hat.

Bezüglich des Energieverbrauchs und des Zeitverhaltens bestätigen sich hier die Ergebnisse des vorhergehenden Abschnitts. Die Abbildungen 7.11, 7.12 und 7.13 zeigen das Laufzeitverhalten der hier verwendeten Videosequenzen für ausgewählte Zeitschranken. Der Energieverbrauch ist hierbei auf den Beginn der Arbeitsphase ab Frame 36 bzw. 41 normalisiert. Bei der Betrachtung des Energieverbrauchs (Abbildung 7.11(a), 7.12(a), 7.13(a)) fällt auf, dass sich ACC und S-ACC nun hinsichtlich ihres Energieverbrauchs deutlich stärker unterscheiden als beim Einsatz in einer absolut regelmäßigen Anwendung. Ebenso ist es hier nicht mehr möglich, Effizienzübergänge zu identifizieren, bei denen RTI schlechter abschneidet als ACC/S-ACC. Diese beiden Feststellungen lassen sich auf die schwankenden Ausführungszeiten der Iterationen innerhalb einer Klasse zurückführen. Da ACC als Komplexität einer Klasse das Maximum der Einzelkomplexitäten annimmt, stellt es hier nun oft mehr als die benötigte Rechenzeit zur Verfügung. Damit werden die Ergebnisse einzelner Iterationen verfrüht fertiggestellt. Die überschüssige Zeit bis zur Zeitschranke kann S-ACC energieeffizient im Schlafmodus überbrücken. Der fehlende Effizienzübergang erklärt sich aus ähnlichen Gründen: wie bereits im vorhergehenden Abschnitt 7.2.2 erläutert, setzt sich das Gesamtverhalten aus den Einzelverhalten der verschiedenen Iterationen zusammen, wobei je nach Zeitschranke unterschiedliche Optimierungsmethoden besser abschneiden. Dadurch, dass ACC nun tendenziell zu viel Rechenzeit zur Verfügung stellt, kann hier RTI immer den geringeren Energieverbrauch erzielen und wird nur noch durch R-ACC übertroffen. Dieser Sachverhalt lässt sich auch gut am Zeitverhalten in den Abbildungen 7.11(b), 7.12(b), 7.13(b) erkennen. Während der Lernphase bis Frame 40 weisen alle Methoden, wie zu erwarten, das gleiche Verhalten auf. In der Arbeitsphase aber zeigt sich nun, dass zwar die Zeitschranken im Allgemeinen eingehalten werden, im Gegensatz zu der optimalen Anwendung findet hier aber bei ACC kein Einpegeln auf die Zeitschranke statt. Stattdessen beenden die meisten mit ACC optimierten Iterationen ihre Arbeit deutlich vor der Zeitschranke.

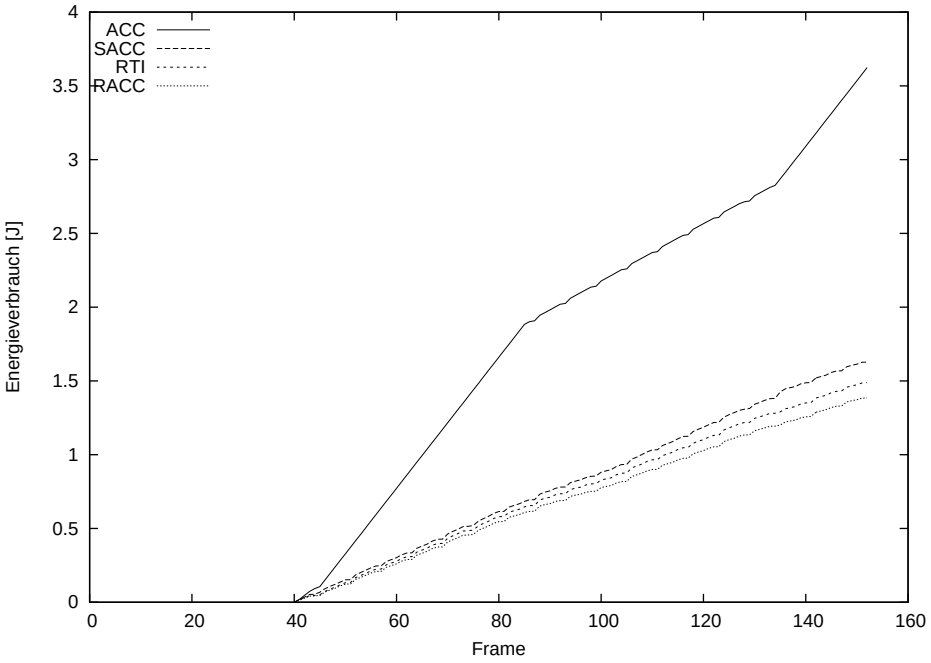


(a) Energieverbrauch SB, IL=40

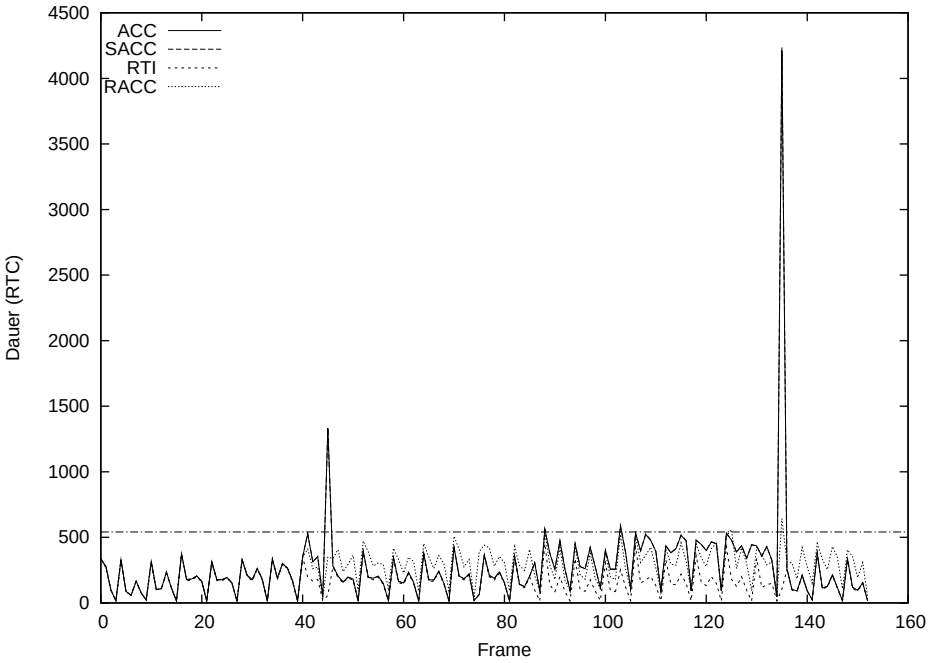


(b) Zeitverhalten SB, IL=40

Abbildung 7.11: Laufzeitverhalten bei der Dekodierung der Videosequenz SB

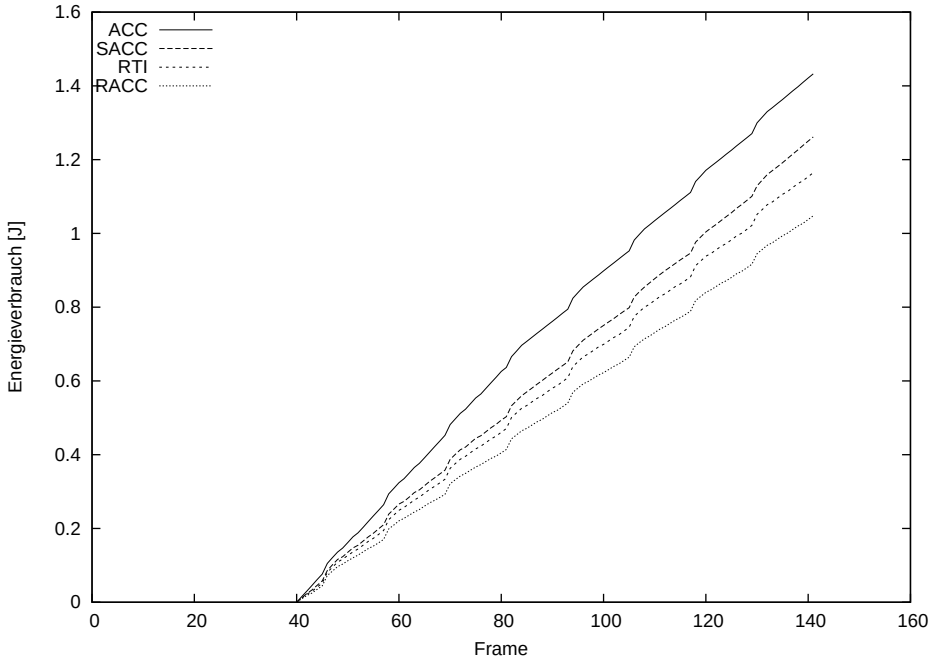


(a) Energieverbrauch WB, IL=40

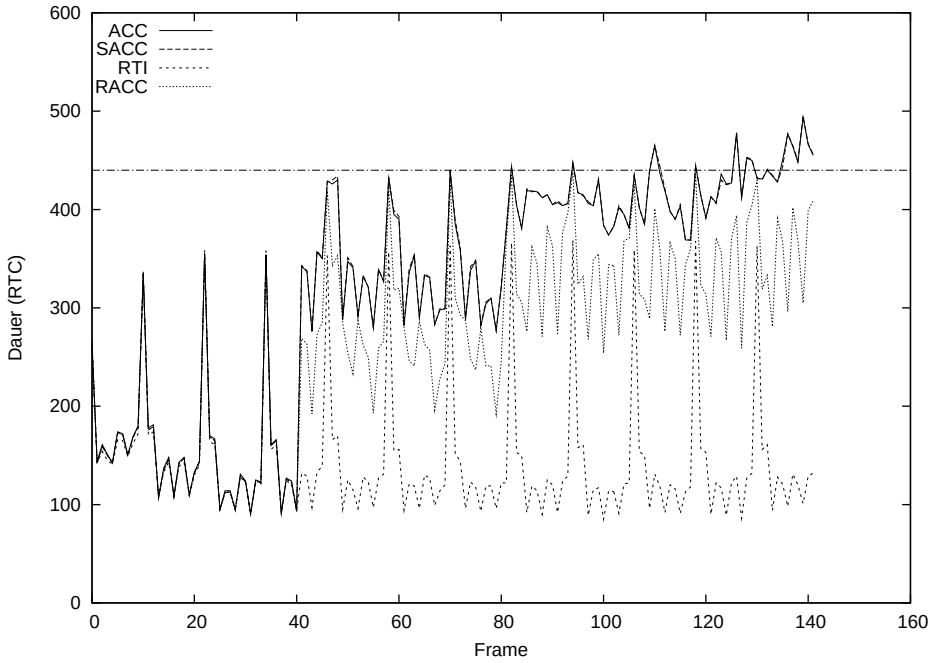


(b) Zeitverhalten WB, IL=40

Abbildung 7.12: Laufzeitverhalten bei der Dekodierung der Videosequenz WB



(a) Energieverbrauch PS, IL=40



(b) Zeitverhalten PS, IL=40

Abbildung 7.13: Laufzeitverhalten bei der Dekodierung der Videosequenz PS

R-ACC erzielt bei allen Evaluierungen den geringsten Energieverbrauch. Dies liegt zum einen daran, dass es wie RTI lange Schlafphasen erzeugt. Zum anderen aber nutzt es die Zeiten, in denen der Prozessor arbeitet, möglichst vollständig aus, was bei ACC/S-ACC nicht der Fall ist. Diese beiden Methoden setzen das Instruction Quantum so, das die Zeitschranke in der vorgegebenen Zeit gerade eingehalten würde. Sobald der Prozessor dieses Instruction Quantum in einer Runde abgearbeitet hat, wird der zugehörige Thread bis zum Beginn der nächsten Schedulingrunde nicht mehr ausgeführt. Damit ergibt sich zumeist noch ein gewisser Leerlauf am Ende jeder Runde, der allerdings durch andere Anwendungen genutzt werden kann.

Wie aus Abbildung 7.12(b) hervorgeht, überschreitet die mit ACC und S-ACC optimierte Anwendung bei der vorliegenden Videosequenz zweimalig extrem ihre Zeitschranke bei der Dekodierung der Frames 46 und 136. In solchen Fällen geht der ACC-Algorithmus davon aus, dass die gelernten Verhaltensparameter falsch sind und wechselt zurück in die Lernphase. Dies schlägt sich insbesondere im Energieverbrauchsdiagramm 7.12(a) in der Kurve von ACC nieder, die an den entsprechenden Stellen einen steilen Anstieg verzeichnet. Sobald ACC einen neue Arbeitsphase einleitet, fällt auch der Anstieg des Energieverbrauchs deutlich flacher aus.

7.2.4 Aufwand

Ein wichtiges Kriterium für den Einsatz der Scheduling-Adaption ist der damit eingeführte zusätzliche Rechenaufwand. Nur wenn dieser in einem vertretbaren Verhältnis zum Rechenaufwand der optimierten Anwendung steht, ist ein Einsatz auch sinnvoll. Grundsätzlich besteht Algorithmus 6.4 aus drei Teilen, die relevant für seinen Rechenaufwand sind:

- **Berechnen und Aufzeichnen der Laufzeit der vorangegangenen Iteration**: Dieser Teil wird bei jedem Aufruf ausgeführt. Er hat einen konstanten Rechenaufwand von ca. 250 Takten.
- Das **Berechnen der Periodizität** erfolgt regelmäßig, aber nicht bei jedem Aufruf. Die Ausführungshäufigkeit und auch die Berechnungsdauer sind abhängig von der Länge des Beobachtungsfensters. Den größten Anteil hat hierbei die Berechnung der zyklischen Autokorrelation, deren Aufwand in $O(n^2)$ liegt, wobei n für die Länge des Beobachtungsfensters steht. Tabelle 7.3 führt die Laufzeiten für verschiedene Längen des Beobachtungsfensters auf.

- Die **Vorhersage des kommenden Rechenaufwands** erfolgt bei jedem Aufruf, sofern sich das Adaptionsmodul nicht gerade in der Lernphase befindet. Die Laufzeit hierfür ist konstant und beträgt ca. 700 Takte.

Tabelle 7.3: Laufzeiten der Periodizitätsberechnung für verschiedene Längen des Beobachtungsfensters

Länge	15	20	25	30
Takte	4699	7267	11119	14953

Insgesamt beträgt die Laufzeit eines Adaptionsschritts in den vorliegenden Beispielen zwischen 5650 und 16000 Takte. Bei diesen Daten handelt es sich um minimal erreichbare Laufzeiten, wenn der Adaption die maximale verfügbare Rechenleistung zur Verfügung steht. Falls andere Programme parallel ablaufen, oder das Instruction Quantum niedriger als das mögliche Maximum ist, so können sich diese Laufzeiten entsprechend verlängern. Dem gegenüber stehen die Laufzeiten der hier verwendeten Programmbeispiele. Bei den Matrixmultiplikationen dauert eine Iteration minimal zwischen 10000 und 685000 Takten (siehe Tabelle 7.1), die Dekodierung der MPEG-Videos benötigt zwischen einer Million und 20 Millionen Takte pro Frame.

Zur weiteren Einschätzung der Kosten von ACC wurde der zusätzliche Energieverbrauch durch die ACC-Instrumentierung bestimmt. Grundlage ist dabei der Energieverbrauch einer durch RTI optimierten Anwendung. Innerhalb der RTI-Instrumentierungen wurden zusätzlich die ACC-Berechnungen ausgeführt, ohne aber deren Ergebnisse auf das laufende Programm anzuwenden. Dabei zeigt sich, dass der zusätzliche Energieverbrauch durch die ACC-Instrumentierung verschwindend gering ist. Bei der Untersuchung eines optimalen Programms (Abschnitt 7.2.2) liegt er zwischen 0,001% und 0,3% des Gesamtenergieverbrauchs. Bei der Optimierung der MPEG-Dekodierung (Abschnitt 7.2.3) liegt er bei ca. 0,004%.

7.2.5 Übertragbarkeit der Ergebnisse auf andere Prozessoren

Die Schedulingadaption mit ACC kann prinzipiell auch in anderen Prozessoren eingesetzt werden, um den Energieverbrauch und die verfügbare Rechenzeit zu optimieren. Voraussetzung hierfür ist, dass auf dem Prozessor ein ähnliches Schedulingverfahren wie PIQ verfügbar ist. Dieses muss nicht unbedingt in Hardware implementiert sein, es kann auch vom Betriebssystem zur Verfügung gestellt werden. Beim Einsatz eines Software-Schedulers ist aber zu beachten, dass ein

Kontextwechsel zwischen zwei Anwendungen deutlich mehr Prozessorzeit in Anspruch nimmt, als dies bei dem hier eingesetzten Hardware-Scheduler der Fall ist. Deshalb werden die Gewinne in einem solchen System möglicherweise niedriger ausfallen.

7.2.6 Mehrfädige Programmausführung

Wie die vorhergehenden Evaluierungen zeigen, verursacht ACC unter den untersuchten Methoden den höchsten Energieverbrauch. Für einfädige Prozessoren ist deshalb eine der Methoden S-ACC, R-ACC oder RTI vorzuziehen, da diese zusätzlich den energieeffizienten Schlafmodus eines Prozessors nutzen. Falls hingegen mehrere Echtzeitanwendungen mit unterschiedlichen Zeitschranken parallel auf einem Prozessor laufen, ist die Nutzung von Schlafphasen, wenn überhaupt, nur noch eingeschränkt möglich, da deren Dauer auf die Laufzeiten und Zeitschranken aller Anwendungen abgestimmt werden müsste. Hier empfiehlt sich stattdessen der Einsatz von ACC, das ohne die Schlafphasen auskommt. Die ACC-Instrumentierung selbst passt dazu jeweils nur das Instruction Quantum der Anwendungen an. Die Anpassung der Taktfrequenz erfolgt dann, wie in Abschnitt 6.1.4 erläutert, über einen zusätzlichen Helper Thread.

7.2.7 Fazit

Eine Vorhersage der Rechenlast von periodischen weichen Echtzeitanwendungen mithilfe von *Autocorrelation Clustering* ist möglich. ACC ist nicht darauf angewiesen, dass zusammengehörende Iterationen exakt dieselbe Rechenlast erzeugen. Allerdings dürfen die Lasten auch nicht zu stark voneinander abweichen, da es ansonsten zu Fehlvorhersagen kommen kann. Die beschriebenen Untersuchungen zeigen, dass sich durch eine möglichst exakte Anpassung der Taktfrequenz gerade dann hohe Energieeinsparungen erzielen lassen, wenn durch geeignete Wahl der Schedulingparameter die Auslastung des Prozessors sehr hoch wird. In solchen Fällen kann der Energieverbrauch geringer sein, als wenn die Anwendung mit *Race-to-Idle-Scheduling* optimiert wird. Die Variation Race-ACC erzielt von allen untersuchten Methoden immer den geringsten Energieverbrauch, indem sie die Vorhersage der Rechenlast ACC und die optimale Nutzung der verfügbaren Rechenzeit sowie langer Schlafphasen des Prozessors, wie bei RTI, miteinander kombiniert. Wenn allerdings mehrere Echtzeitanwendungen mit unterschiedlichen Zeitschranken parallel auf einem Prozessor laufen, ist die Nutzung der Schlafmodi nicht oder nur mit hohem zusätzlichen Berechnungsaufwand möglich. In diesem Fall stellt ACC eine einfache Möglichkeit dar, den Energieverbrauch zu senken, ohne das Zeitverhalten anderer Anwendungen zu beeinflussen. Wie dies mithilfe

einer übergelagerten Managementkomponente bewerkstelligt werden kann, wird Abschnitt 7.5 zeigen.

7.3 Software Watchdog zur Funktionsüberwachung von Anwendungen

Auch der Software-Watchdog aus Abschnitt 6.2 eignet sich sowohl zum alleinstehenden Betrieb als auch zur Kombination mit DCERT. Auf eine funktionale Evaluierung kann verzichtet werden, da sich dieses Verhalten schon eindeutig aus seiner Definition ergibt. Durch den Vergleich der Heartbeats und der Systemzeit kann er den Ausfall einer Anwendung frühestens dann feststellen, wenn diese ihre Zeitschranke überschritten hat. Die tatsächliche Dauer zwischen dem Überschreiten der Deadline und dem Erkennen des Ausfalls hängt von der Frequenz ab, mit der die Überprüfung durchgeführt wird. Dies liegt vollständig in der Hand des Entwicklers. Die folgende Evaluierung geht deshalb insbesondere auf das Zeitverhalten des Software-Watchdog ein.

7.3.1 Kosten der Instrumentierung

Zusätzlich zu der Instrumentierung Heartbeat_Tick (Algorithmus 6.7) werden noch Funktionen zum Starten (Start_Watchdog) und Anhalten (Stop_Watchdog) des Watchdog benötigt. Auch deren Aufruf muss in der überwachten Anwendung integriert sein. Start_Watchdog muss die Heartbeat-Variablen initialisieren und außerdem dem Watchdog Service mitteilen, dass er diese nun überwachen soll. Bei statischer Allokation der Datenstruktur kann dies durch Setzen eines einzelnen Flags erfolgen. Auf dem CarCore dauert diese Operation 25 Takte. Jeder einzelne Heartbeat_Tick schlägt mit weiteren 13 Takten zu Buche. Das Anhalten des Watchdog Service erfolgt durch Zurücksetzen des mit Start_Watchdog gesetzten Flags. Stop_Watchdog benötigt hierfür 4 Takte.

7.3.2 Kosten des Watchdog Service

Die Kosten für die reine Prüfung durch den Watchdog Service (Heartbeat_Check) belaufen sich auf 60 bis 90 Takte, je nachdem ob die Anwendung eine Deadline verpasst hat und welche Reaktion der Watchdog Service daraufhin wählt. Hinzu kommen noch die Zeiten für das Neustarten des Applikationsthreads oder die anwendungsspezifische Reaktion. Bei einem

Neustart sind hier etwa 3.300 Takte zu veranschlagen. Falls der Watchdog Service mehrere Anwendungen überwacht, also in eine Schleife eingebettet ist, erhöhen sich die genannten Kosten auf 75 bis 110 Takte für jede überwachte Anwendung, da der Zugriff auf die Daten hier geringfügig aufwändiger ist. Dazu kommen noch Kosten für die Verwaltung der Schleife selbst, die sich auf etwa 15 Takte pro Iteration belaufen.

7.3.3 Fazit

Die Watchdog-Instrumentierung bringt nur einen minimalen Overhead in die Anwendung. Tatsächlich ist dieser sogar geringer als beim Einsatz eines Hardware-Watchdog, auf den üblicherweise nur mittels aufwändiger System Calls über das Betriebssystems zugegriffen werden kann. Auch der Watchdog Service selbst erzeugt nur geringe Zusatzkosten bezüglich der Laufzeit. Nur in Fehlersituationen benötigt er merklich Rechenzeit für die entsprechende Reaktion. Aber auch hier ist der Neustart des Anwendungsthreads deutlich günstiger als ein Neustart der gesamten Knotensoftware.

7.4 DCERT: Szenario

Die Zusammenarbeit von DCERT mit den verschiedenen Modulmanagern wurde an mehreren Szenarien evaluiert. Alle diese Evaluierungen basieren auf einer einheitlichen Systemkonfiguration bezüglich der Statusparameter, Monitore und Aktoren, die der folgende Abschnitt einführt. Die zusätzlichen Konfigurationen auf Anwendungsebene finden sich dann in den Abschnitten 7.5 und 7.6 der einzelnen Szenarien.

7.4.1 Statusparameter

Die Evaluierungen der folgenden Abschnitte verwenden als Statusparameter die Menge

$$\begin{aligned}\Pi \;=\; \{\,&\text{FMAX, FMIN, HIBATT, EMIG\_STATE\_}x\text{, MISSDL,}\\ &\text{LOBATT, HIPOWC, LOPERF, IMMIG\_STATE\_}x\text{, HIPERF}\,\}.\end{aligned}$$

Die Triggerparameter sind durch die Menge

$$\begin{aligned}T \;=\; \{\,&\text{MISSDL, LOBATT, HIPOWC, LOPERF,}\\ &\text{IMMIG\_STATE\_}x\text{, HIPERF}\,\}\end{aligned}$$

definiert. Die Statusparameter sind auf Basis von 64-Bit-Integern implementiert. Sie haben die folgenden Bedeutungen:

FMAX Die Taktfrequenz ist momentan auf ihrem Maximalwert. Sie kann also nicht weiter erhöht werden.

FMIN Die Taktfrequenz ist momentan minimal, sie also nicht mehr weiter gesenkt werden kann.

HIBATT Die Batterie ist noch fast voll (>75% der Kapazität).

EMIG_STATE_x Der Migrationsslot $x \in [0 \dots n-1]$ enthält eine migrierbare Anwendung.

MISSDL weist darauf hin, das Anwendungen ihre Zeitschranken verfehlt haben *(Trigger, Gewicht* 2^{63}*)*.

LOBATT Die in der Batterie verbleibende Energie ist sehr gering (<25% der Gesamtkapazität). *(Trigger, Gewicht* 2^{59}*)*.

HIPOWC Aktuell ist der Energieverbrauch sehr hoch. Dies hat einen negativen Einfluss auf die Lebensdauer der Batterie *(Trigger, Gewicht* 2^{58}*)*.

LOPERF Die aktuelle Performanz des Prozessors reicht nicht für die angeforderte Rechenleistung aus *(Trigger, Gewicht* 2^{57}*)*.

IMMIG_STATE_x In Migrationsslot x wartet eine Anwendung auf Migrationserlaubnis *(Trigger, Gewicht* $2^{40} - 2^{47}$*)*.

HIPERF Die momentan angeforderte Rechenleistung nutzt die bereitgestellte Performanz nur unvollständig aus *(Trigger, Gewicht* 2^{3}*)*.

Diese Statusmeldungen liefern ein vereinfachtes Abbild des jeweils aktuellen Systemzustands. Sie werden bei Bedarf von den Monitoren im nächsten Abschnitt erzeugt. Entsprechend markierte Parameter dienen gleichzeitig als Triggerparameter (siehe Abschnitt 5.1.1 und Definition 5.4). Tritt ein solcher Parameter als Nachricht auf, so löst er damit automatisch einen Klassifizierungszyklus von DCERT aus.

7.4.2 Monitore

Das Basissystem aus CAROS und den vorgestellten Modulmanagern verfügt über die folgenden Monitore:

- Der **Frequenzmonitor** überwacht die Taktfrequenz und sendet gegebenenfalls Nachrichten FMAX oder FMIN.

- Der **Batteriemonitor** überwacht die Batterie des Systems. Bezüglich des Ladezustands kann er die Nachrichten HIBATT oder LOBATT senden. Eine hohe Leistungsentnahme zeigt er mit der Nachricht HIPOWC an.
- Der **Migrationsmonitor** überwacht die Anwendungsslots des Migrationsdienstes. Falls migrierbare Anwendungen laufen, sendet er EMIG_STATE_x. Falls ein anderer Host eine Anwendung auf den eigenen Host verschieben möchte, fordert er durch IMMIG_STATE_x eine Bestätigung durch DCERT an. Diese Zustandswerte werden durch die anderen Operationen des Migrationsdienstes gesetzt, so dass im Monitor kein weiterer Rechenaufwand entsteht.
- Der **Schedulingmonitor** überwacht das Zeitverhalten aller laufenden Threads (siehe Abschnitt 6.3). Falls am Ende jeder Schedulingrunde noch viel freie Takte verbleiben, sendet er die Nachricht HIPERF. Kann dagegen das Instruction Quantum wenigstens eines Threads nicht ausreichend befriedigt werden, so zeigt er dies durch LOPERF an.
- Ein **Watchdog-Monitor** überwacht das Zeitverhalten einzelner Anwendungen. Falls eine Anwendung wiederholt ihre Zeitschranke verpasst hat, meldet er dies mit MISSDL an DCERT.

7.4.3 Aktore

Zur Fehlerbeseitigung durch DCERT stellen CAROS und die Modulmanager die Aktorenmenge

$$\begin{aligned}\mathcal{A} \;=\; \{\,&\text{IncFreqActor, DecFreqActor, EmigrationActor,}\\ &\text{ImmigrationActor(1), ImmigrationActor(2)}\,\}\end{aligned}$$

zur Verfügung. Diese Aktoren sind wie folgt definiert:

IncFreqActor Vorbedingung: ¬FMAX, HIBATT,
Trigger: LOPERF, MISSDL
Die Erhöhung der Taktfrequenz ist nur dann möglich, wenn der Knoten über ausreichende Energiereserven verfügt und noch nicht mit maximaler Frequenz arbeitet. Sie wird durch ein Performanzprobleme ausgelöst.

DecFreqActor Vorbedingung: ¬FMIN,
Trigger: HIPOWC, LOBATT, HIPERF
Verfügt der Knoten über einen Leistungsüberschuss, so sollte die Taktfrequenz wenn möglich gesenkt werden. Ebenfalls eignet sich dieser Aktor, um eine für die Batterie schädliche hohe Leistungsaufnahme zu reduzieren.

EmigrationActor Vorbedingung: EMIG_STATE_x,
Trigger: LOPERF, MISSDL
Der Emigrationsaktor sendet beim Aufruf eine Migrationsanfrage an alle ihm bekannten Knoten im Netz.

ImmigrationActor (1) Vorbedingung: ¬LOPERF,
Trigger: IMMIG_STATE_x
Der Immigrationsaktor sendet beim Aufruf durch DCERT eine Migrationserlaubnis an den Quellknoten der Anfrage, an die er gebunden wurde. Diese Variante kann nur aufgerufen werden, wenn der Knoten keine Performanzprobleme hat.

ImmigrationActor (2) Vorbedingung: ¬FMAX, HIBATT, LOPERF,
Trigger: IMMIG_STATE_x
Dieser Immigrationsaktor verhält sich ebenso wie der vorherige, unterscheidet sich allerdings in der Vorbedingung. DCERT kann ihn auch dann aufrufen, wenn der Knoten aktuell Performanzprobleme hat, aber in der Lage ist, diese Probleme durch die Erhöhung der Taktfrequenz zu beseitigen.

7.5 DCERT mit dem Modulmanager zur Schedulingadaption

Die folgende Evaluierung setzt die Integration von DCERT mit ACC (Abschnitt 6.1.4) um. Dabei nutzt sie auch den Schedulingmonitor aus Abschnitt 6.3. Als Anwendung dient die Dekodierung von MPEG-Videos, wie sie bereits in Abschnitt 7.2.3 zum Einsatz kam. Die ACC-Instrumentierung passt in diesem Fall nur noch das Instruction Quantum des dekodierenden Threads an die vorhergesagte Rechenlast an, nicht aber die Taktfrequenz des Prozessors. Trotzdem fließt das Instruction Quantum noch indirekt in die Wahl der Taktfrequenz ein. Der Schedulingmonitor überwacht das Zeitverhalten des Schedulers, also die Zahl der Freitakte am Ende einer Runde beziehungsweise die Anzahl der Instruktionen eines PIQ-Threads, die bis Rundenende nicht ausgeführt werden konnten. Er besitzt also ein genaues Bild über die *tatsächliche* Lage der Performanz, welches er DCERT in abstrahierter Form zur Verfügung stellt. DCERT kann dann aufgrund dieser Informationen die Taktfrequenz an die tatsächliche Lage angepasst verändern.

7.5.1 Ergebnisse

Das Verhalten der Kombination aus DCERT und ACC wurde anhand der Dekodierung von MPEG-Videosequenzen untersucht, wie sie auch in Abschnitt 7.2.3 zum Einsatz kamen. Die Abbildungen 7.14, 7.15 und 7.16 zeigen den Energieverbrauch sowie das Zeitverhalten dieser Sequenzen. Zur besseren Vergleichbarkeit finden sich hier auch die entsprechenden Werte für die reine ACC-Instrumentierung. Der Energieverbrauch liegt in allen Fällen höher als bei der reinen ACC-Instrumentierung. Die Ursache hierfür liegt in der Art und Weise, wie DCERT die Taktfrequenz wählt.

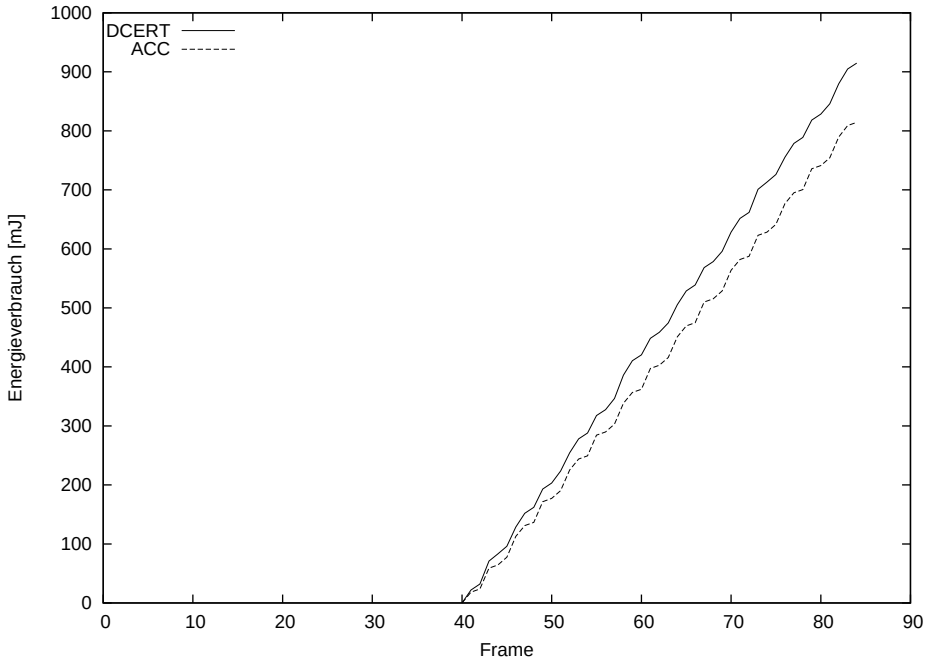
Abbildung 7.17 vergleicht die Taktfrequenz, wie sie beim Einsatz von ACC mit direkter Frequenzanpassung und über DCERT verläuft. Man sieht, dass ACC für jede Iteration, also im vorliegenden Fall die Decodierung eines Frames, genau eine Taktfrequenz auswählt. DCERT hingegen variiert die Taktfrequenz stärker und führt die Anpassungen auch stärker abgestuft durch. Es fällt aber auch auf, dass der Prozessor bei der Optimierung durch DCERT deutlich mehr Zeit auf hohen Taktfrequenzen verbringt. Dies schlägt sich auch entsprechend im Energieverbrauch (Abbildungen 7.14(a), 7.16(a)) nieder.

Gleichzeitig ändert sich auch das Zeitverhalten der optimierten Anwendung (Abbildungen 7.14(b), 7.16(b)). Sie erreicht ihre Zeitschranken zumeist mit deutlich mehr Spielraum, nur bei den sehr aufwändigen I-Frames unterscheidet sich die Zeitdauer der Berechnung nahezu nicht.

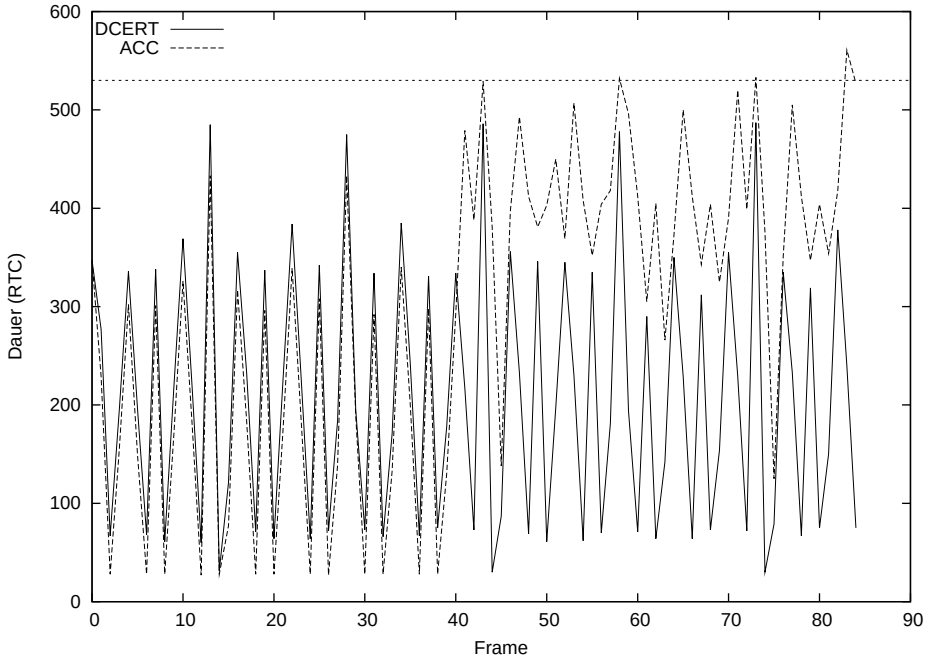
Eine Sonderstellung nimmt hier die Dekodierung der Videosequenz WB (Abbildungen 7.15(a) und 7.15(b)) ein. Bei der Optimierung allein durch ACC kommt es hier an zwei Stellen bei Frame 46 und 136 zu einem so extremen Verpassen der Zeitschranke, dass ACC zurück in die Lernphase wechselt. Entsprechend schlägt sich dies auch im Energieverbrauch nieder, der sich in diesen Phasen entsprechend stärker erhöht. Durch Kombination mit DCERT verpasst die Anwendung hingegen keine Zeitschranken. Dadurch werden die energieaufwändigen Lernphasen vollständig vermieden, wodurch sich insgesamt ein geringerer Energieverbrauch ergibt.

7.5.2 Fazit

In Fällen, in denen ACC bereits alleinstehend zufriedenstellend arbeitet, führt die Kombination mit DCERT eher zu einem höheren Energieverbrauch. Es zeigt sich aber auch, dass durch die Kombination mit DCERT jegliches Verpassen von Zeitschranken vermieden wird. Insofern ensteht hier auch ein gewisser Gewinn bezüglich der Ausführungsqualität. Ein deutlicher Gewinn entsteht allerdings, wenn die Optimierung mit ACC allein dazu führt, dass die Anwendung gelegent-

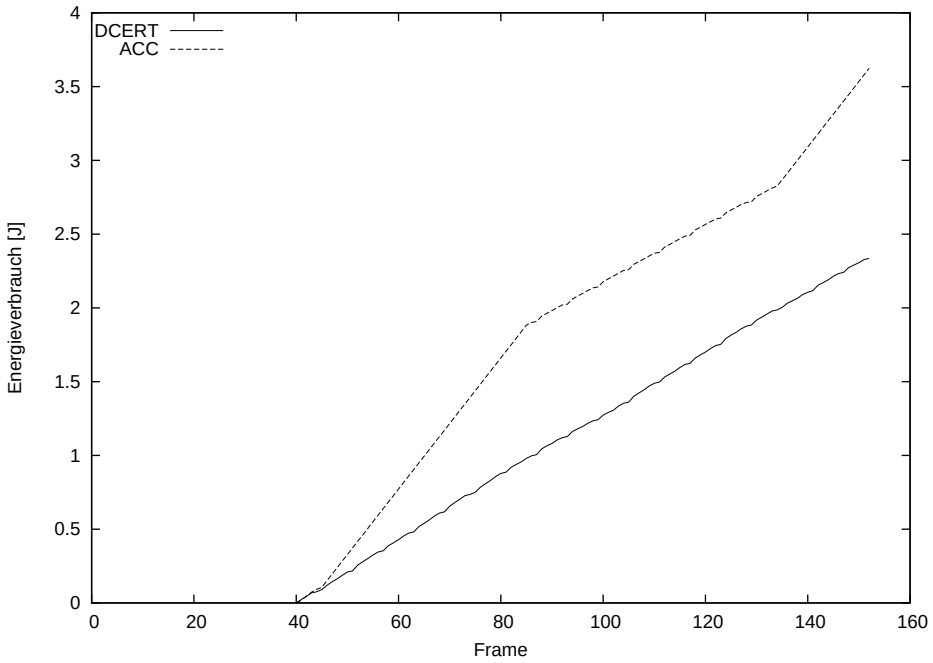


(a) Energieverbrauch SB/DCERT, IL=40

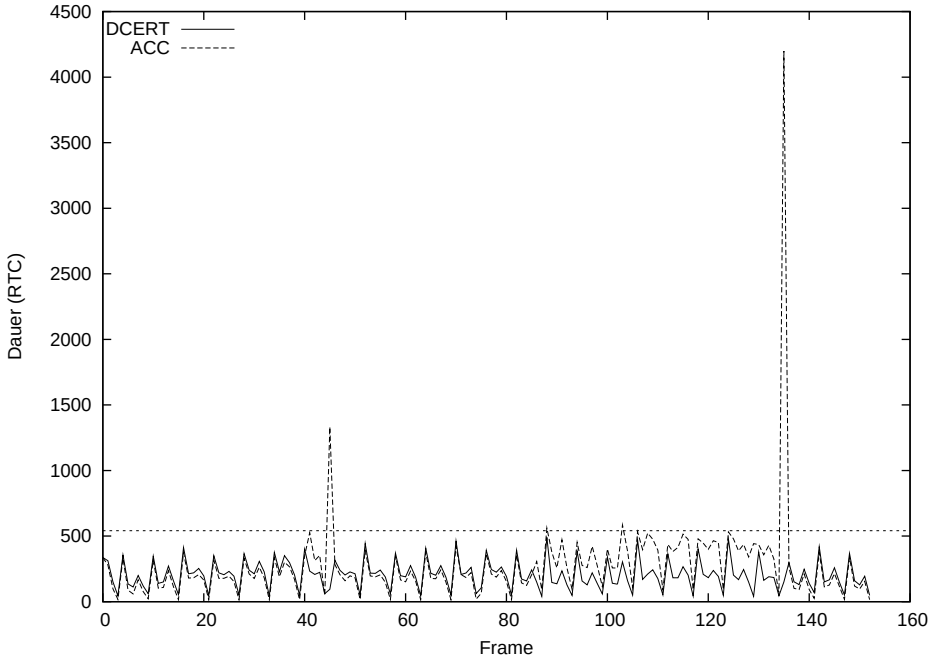


(b) Zeitverhalten SB/DCERT, IL=40

Abbildung 7.14: Laufzeitverhalten bei der Dekodierung der Videosequenz SB, Energieoptimierung durch ACC und DCERT

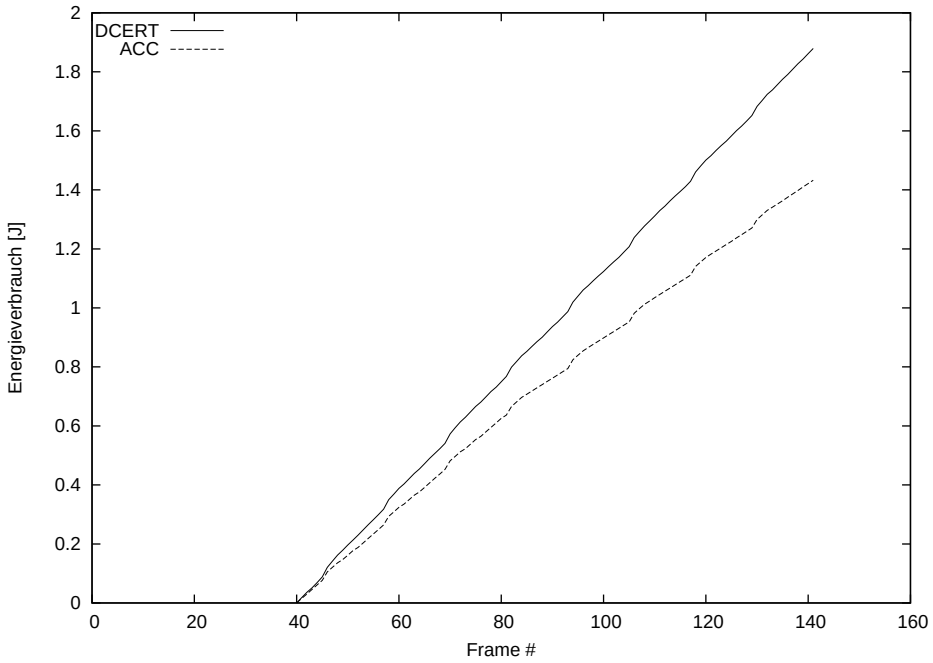


(a) Energieverbrauch WB/DCERT, IL=40

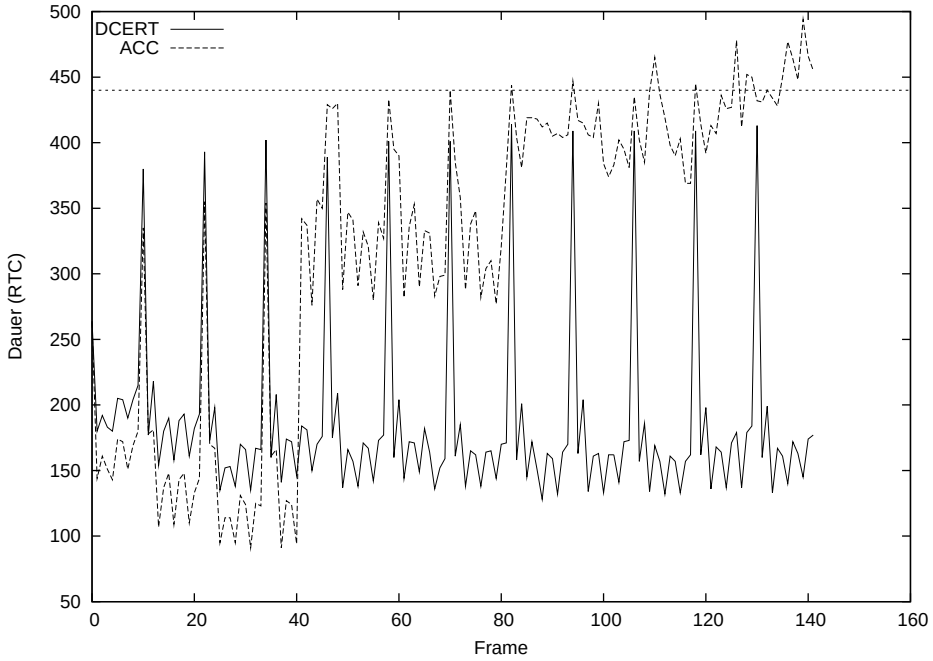


(b) Zeitverhalten WB/DCERT, IL=40

Abbildung 7.15: Laufzeitverhalten bei der Dekodierung der Videosequenz WB, Energieoptimierung durch ACC und DCERT



(a) Energieverbrauch PS/DCERT, IL=40



(b) Zeitverhalten PS/DCERT, IL=40

Abbildung 7.16: Laufzeitverhalten bei der Dekodierung der Videosequenz PS, Energieoptimierung durch ACC und DCERT

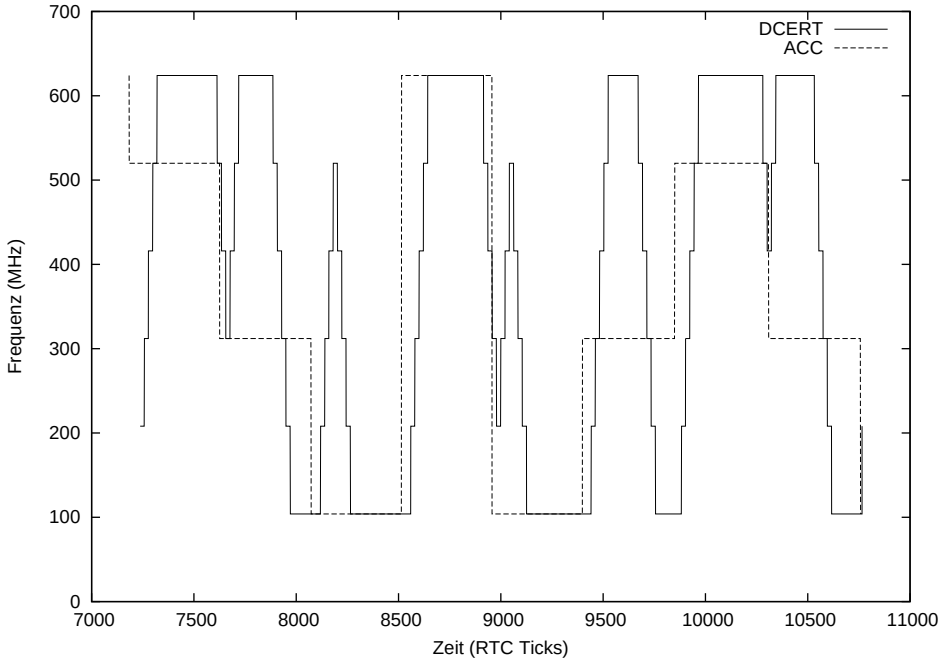


Abbildung 7.17: Frequenzverlauf während der MPEG-Dekodierung von ACC im Vergleich zu ACC mit DCERT (Ausschnitt)

lich ihre Zeitschranken sehr weit verpasst. Dieses Verhalten wird durch DCERT eliminiert. Die zusätzliche Beobachtung des Scheduler-Verhaltens durch DCERT ermöglicht hier eine flexiblere Anpassung der Taktfrequenz. Dies wirkt sich positiv auf das Zeitverhalten der Anwendung und letztendlich auch auf den Energieverbrauch aus.

7.6 DCERT mit dem Modulmanager zur Anwendungsmigration

Das Basiszenario für diese Evaluierung umfasst einen Ursprungsknoten, auf dem eine migrierbare Anwendung läuft sowie einen oder mehrere mögliche Zielknoten, die nahezu frei von Rechenlast sind und somit weitere Anwendungen aufnehmen können.

DCERT kann die Migration einer Anwendung aus mehreren Gründen auslösen. Unmittelbar ergibt sich eine solche Reaktion aber immer dann, wenn der Knoten überlastet ist und DCERT infolgedessen die Statusmeldung LOPERF vom Schedulingmonitor empfängt. Dabei löst es eine Migration nicht sofort aus, sondern versucht erst mit einfacheren Aktoren dem Missstand entgegenzuwirken. In

dem hier verwendeten Szenario nutzt es dazu den *IncFreqActor*, um die Taktfrequenz zu erhöhen. Erst wenn dieser Aktor nicht mehr zur Verfügung steht, wählt DCERT die Anwendungsmigration als Reaktion aus. Dies ist zum einen der Fall, wenn die Taktfrequenz bereits maximal ist, kann aber auch über die verbleibende Energie der Batterie beeinflusst werden. Falls diese zu gering ist, der Batteriemonitor also gleichzeitig den Zustand LOBATT meldet, schließt der Aktor eine Frequenzerhöhung ebenfalls aus.

Durch den Aufruf des Emigrationsaktors für eine bestimmte Anwendung stößt DCERT nun einen Migrationsvorgang an. Die Ausführung des Aktors selbst ist nur mit geringen Kosten verbunden. Seine Aufgabe ist es, im eigenen Knoten die Migrationsanforderung zu vermerken sowie eine Migrationsfrage an alle ihm bekannten Knoten im Netz zu senden. Alle weiteren Aktionen, die den Migrationsvorgang auf dem Ursprungsknoten betreffen, führt der Migrationsdienst selbstständig innerhalb eines eigenen Threads durch. Der weitere Migrationsvorgang hat also keinen Einfluss mehr auf das Zeitverhalten von DCERT. Der Migrationsdienst selbst arbeitet nur dann, wenn der Knoten eine Migrationsnachricht empfängt.

In einem Netz mit insgesamt n Knoten erfordert eine erfolgreiche Migration zwischen $2n - 1$ und $3n - 3$ Nachrichten. Tabelle 7.4 zeigt, welche Nachrichtentypen wie oft versendet werden. Zunächst versendet der *Emigrationsaktor* je eine Migrationsanfrage an die anderen Knoten im Netz, insgesamt also $n - 1$ Nachrichten. Die anderen Knoten beantworten diese Anfrage dann entweder mit einer Erlaubnis oder Ablehnung, was wiederum zu $n - 1$ Nachrichten führt. Wenn der Ursprungsknoten eine Ablehnung empfängt, führt er mit deren Absender für den Rest des Migrationsvorgangs keine weitere Kommunikation. Aus all den Knoten, die ihm eine Erlaubnis gesendet haben, wählt er einen als Zielknoten aus und sendet diesem die Anwendung (MIGR_MIGRATION). Den anderen Knoten sendet er jeweils eine Abbruchnachricht, ingesamt also bis zu $n - 2$ Nachrichten. Die minimale Nachrichtenzahl $2n - 1$ wird dann erreicht, wenn genau ein Knoten eine Migration erlaubt und alle anderen die Migration ablehnen. Maximal $3n - 3$ Nachrichten entstehen, wenn jeder Knoten eine Migration erlaubt und so der Ursprungsknoten noch $n - 2$ Abbruchnachrichten versenden muss.

Ebenfalls in Tabelle 7.4 ist die Größe der einzelnen Nachrichtentypen. Das Basispaketformat belegt 32 Bytes. Bei einer Migrationsanfrage enthält die Nachricht zusätzlich noch Informationen über die Rechenlast, die die Anwendung erzeugt. Die Größe der eigentlichen Migrationsnachricht wird maßgeblich von der Anwendung bestimmt. Diese Nachricht enthält zum einen den gesamten Anwendungscode, und zum anderen auch die Daten, die die Anwendung auf den Zielknoten mitnehmen muss.

Tabelle 7.4: Anzahl der Nachrichten für einen Migrationsvorgang bei n Knoten; die Größe einer Nachricht vom Typ MIGR_MIGRATION hängt von der Größe der migrierten Anwendung ab.

Nachrichtentyp	Anzahl	Größe
MIGR_REQUEST	$n-1$	36 Bytes
MIGR_PERMISSION	$0 \ldots n-1$	32 Bytes
MIGR_DENIAL	$n-1 \ldots 0$	32 Bytes
MIGR_MIGRATION	$0 \ldots 1$	$32+x$ Bytes
MIGR_CANCEL	$0 \ldots n-2$	32 Bytes

Die Zeitkosten für eine Migration variieren mit der Größe des Netzes sowie mit Umfang und Struktur der Anwendung. Weiteren Einfluss hat hier auch die zugrundeliegende Übertragungstechnik. Die Ausführungszeit des *Emigrationsaktors* hängt von der Anzahl der zu versendenden Nachrichten ab. Abbildung 7.18 zeigt, dass diese Ausführungszeit linear mit der Anzahl der Knoten steigt. Die mittleren Kosten pro versendeter Nachricht betragen dabei 1038 Takte. Unabhängig von der Anzahl der Nachrichten entstehen beim Aufruf des Emigrationsaktors mittlere Fixkosten von 1208 Takten für das Vorbereiten der Nachricht.

Die Verarbeitung der Migrationsanforderung auf dem möglichen Zielknoten hängt von der Entscheidung des Migrationsdienstes ab. Falls dieser schon feststellt, dass der Knoten nicht in der Lage ist eine Anwendung aufzunehmen, so beantwortet er die Anfrage direkt mit MIGR_DENIAL. Dies dauert etwa 2000 Takte. Falls er hingegen eine Aufnahme für möglich hält, stellt er eine entsprechende Anfrage an DCERT, wodurch Zeitkosten in Höhe von ca. 5000 Takten entstehen.

Wenn DCERT der Aufnahme einer Anwendung zustimmt, sendet der Migrationsdienst eine entsprechende MIGR_PERMISSION an den Ursprungsknoten. Der dortige Migrationsdienst muss nun entscheiden, ob er die Anwendung an diesen Knoten sendet, oder den Migrationsvorgang bezüglich dieses Knotens abbricht und einen anderen Zielknoten auswählt. Die Zeitkosten für einen Abbruch sind konstant und betragen etwa 2000 Takte. Die Verarbeitung auf Empfängerseite verursacht einen weiteren Aufwand von etwa 2500 Takten. Entscheidet sich der Migrationsdienst hingegen, die Anwendung zu versenden, so setzen sich die Zeitkosten hierfür wie folgt zusammen:

- **Anhalten der Anwendung:** Der Migrationsdienst sendet ein Signal an die Anwendung, die daraufhin selbstständig die Arbeit einstellen und ihre Daten in einen konsistenten Zustand bringen muss. Das Ende dieser Arbeit signalisiert sie wiederum dem Migrationsdienst, der daraufhin mit der Migration fortfährt. Die Zeitdauer für diesen Vorgang hängt nahezu

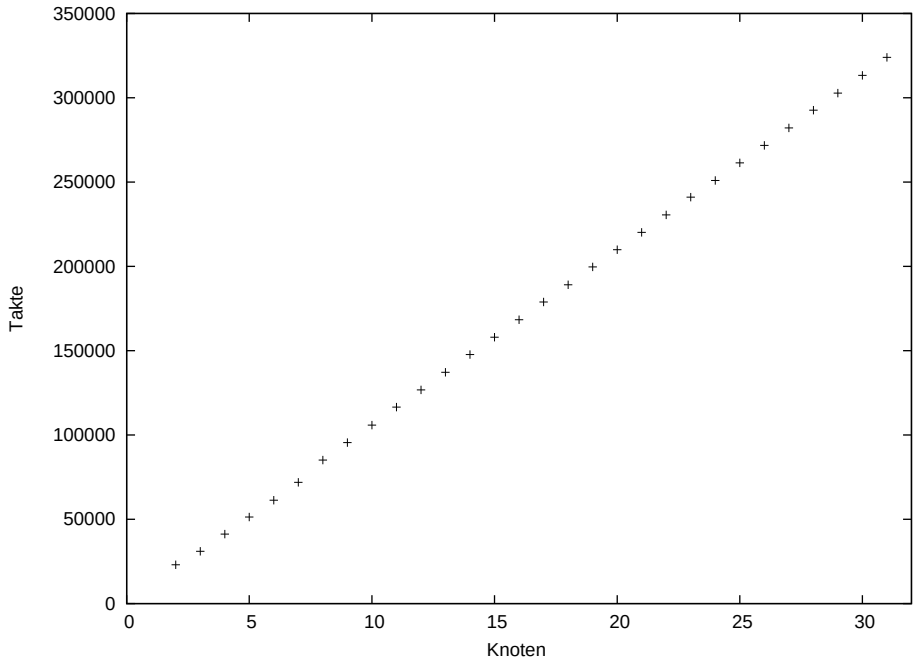


Abbildung 7.18: Ausführungszeit des Emigrationsaktors in Abhängigkeit von der Netzgröße

vollständig von der Anwendung ab, kann aber bei Kenntnis der Anwendung bestimmt werden.

- **Vorbereiten der Migrationsnachricht:** Hier legt der Migrationsdienst den Speicher für die Migrationsnachricht an. Diese enthält den Objektcode der Anwendung sowie die Anwendungsdaten. Die Dauer dieses Schritts wird maßgeblich von der Art der Speicherallokation beeinflusst.
- **Sichern der Anwendungsdaten:** Die Anwendung muss eine Funktion bereitstellen, mit der sie relevante Daten für die Migration in einen vom Migrationsdienst bereitgestellten Speicherbereich sichert. Die Zeit für diese Sicherung hängt wiederum von der Anwendung und insbesondere dem Umfang dieser Daten ab.

Entsprechend hängt auch das Einbinden einer empfangenen Anwendung stark von dieser ab. Die Zeit hierfür setzt sich aus folgenden Komponenten zusammen:

- **Bindungsprozess:** In diesem Schritt erstellt der Runtime Linker aus dem Objektcode der Anwendung ein lauffähiges Speicherabbild. Dabei muss er die Abhängigkeiten sowohl innerhalb der Anwendung als auch zu weiteren Modulen oder dem Betriebssystem auflösen. Der Umfang und die Art dieser hier notwendigen *Relocations* beeinflussen die Zeitdauer des Bindungsprozesses entscheidend und hängen stark von der Struktur der Anwendung ab.

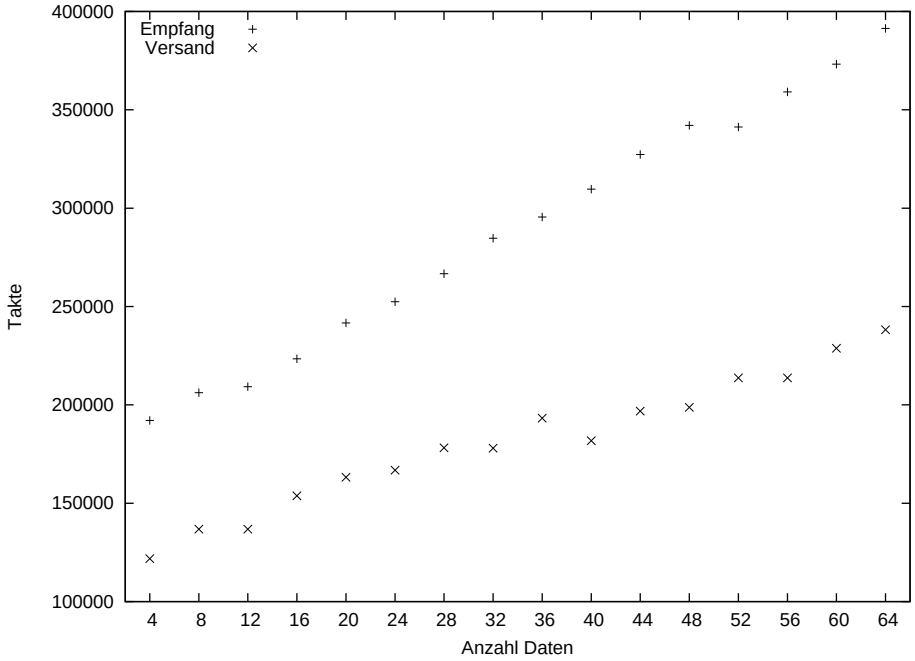


Abbildung 7.19: Ausführungszeit für den Versand und Empfang einer Applikation in Abhängigkeit vom Umfang der Daten

Eine Bestimmung der Zeitdauer ist nur bei Kenntnis der konkreten Anwendung möglich.

- **Wiederherstellen der Anwendungsdaten:** Die Anwendung stellt ihre Datenbereiche aus den vor der Migration gesicherten Daten wieder her. Wie beim Sichern der Anwendungsdaten hängt auch dieser Schritt wiederum vom Umfang der migrierten Daten ab.

Abbildung 7.19 zeigt die jeweilige Dauer des Versand- und Empfangsvorgangs einer Anwendung in Abhängigkeit vom Umfang der Anwendungsdaten. Jedes Datum entspricht dabei einem 32-Bit-Wert. Die untersuchte Anwendung stellt dabei ausschließlich die für die Migration notwendigen Funktionalitäten zur Verfügung. Beim Bindungsprozess muss der Linker dazu für jedes Datum vier Relocations auflösen, je zwei für das Serialisieren und Deserialisieren der Daten. Hinzu kommen noch die Kosten für das Wiederherstellen der Daten. Beim Versand entstehen nur die Variablen Kosten für das Sichern der Daten. Wie man in der Abbildung erkennt, steigen die Kosten für den Empfang einer Anwendung entsprechend stärker als beim Versand. Ebenso sieht man, dass der Empfang an sich aufgrund des Bindungsprozesses mit einem deutlich höherem Aufwand verbunden ist.

7.6.1 Fazit

Die Möglichkeit zum Verschieben von Anwendungen zwischen einzelnen Knoten ist essentiell, um die Flexibilität eines eingebetteten Systems zu erhöhen. Wie man sieht, sind dabei schon die Basisfunktionalitäten mit einem erheblichen Zeitaufwand verbunden. Hinzu kommt in allen Fällen noch der Aufwand für den eigentlichen Bindungsprozess. Selbst bei sehr einfachen Programmen dauert dieser mindesten 100000 Takte. Dieser Sachverhalt unterstreicht damit die bereits in Kapitel 2 formulierte Forderung nach zeitlich isolierten *Helper Threads*. Nur so ist es möglich, auf einer Steuereinheit einen Migrationsprozess durchzuführen, ohne dabei eine parallel arbeitende Echtzeitanwendung zu beeinflussen. Dies wird gerade auch deshalb relevant, da man bei der Entwicklung der Steuereinheit unter Umständen noch gar nicht absehen kann, welche Anwendungen zusätzlich auf diese migriert werden. Zu diesem Zeitpunkt ist also der Zeitaufwand eines möglichen Migrationsprozesses in weiten Teilen unvorhersehbar. Durch dessen zeitliche Isolierung in einem Helper Thread kann aber sichergestellt werden, dass er keine Auswirkungen auf knotenspezifische Echtzeitanwendungen hat.

7.7 Kosten durch DCERT

Grundsätzlich lässt sich jeder Durchlauf von DCERT in drei Phasen unterteilen:

1. **Monitoring:** DCERT sammelt von allen registrierten Monitoren deren aktuelle Zustandswerte ein.
2. **Klassifizierung:** DCERT vergleicht den aktuellen Systemzustand mit den Vorbedingungen und Triggermengen der registrierten Aktoren. Es wählt jene Aktoren aus, die für den aktuellen Systemzustand eine Verbesserung erzielen können.
3. **Reaktion:** Aus den verfügbaren Aktoren wählt DCERT einen zur Reaktion aus und führt diesen aus.

Die im Folgenden präsentierten Laufzeiten beziehen sich auf die Ausführung von DCERT als DTS-Thread. Insofern geben sie minimal erreichbare Laufzeiten an. DCERT selbst läuft gewöhnlich als Helper Thread im Zeitschatten der eigentlichen Anwendungen des Knotens. Dazu führt man DCERT lediglich als PIQ-Thread aus. Die tatsächlichen Laufzeiten können daher auch höher sein als die hier angegebenen Zahlen.

Die Basiskosten für DCERT belaufen sich auf 260 Takte. Diese entstehen dann, wenn keinerlei Monitore registriert sind, DCERT also auch keine kritischen Sy-

Tabelle 7.5: Kosten für die Ausführung von Monitoren

Monitor	Kosten (Takte)
Frequenzmonitor	131 ± 4
Batteriemonitor	167 ± 3
Migrationsmonitor	124
Schedulingmonitor	204 ± 2
Watchdog-Monitor	124

stemzustände erkennt. Eine Klassifizierung ist dann nicht möglich, eine Reaktion ebenfalls nicht.

7.7.1 Ergebnisse

Monitoring Die Kosten für die Bestimmung des aktuellen Systemzustands setzen sich folgendermaßen zusammen:

- **Fixkosten von DCERT:** Diese Kosten fallen unabhängig von der Anzahl der Monitore und dem aktuellen Systemzustand an. In der verwendeten Implementierung betragen sie 154 Takte.
- **Fixkosten pro Monitor:** Für jeden Monitor, den DCERT abfragen muss, entstehen Zusatzkosten von 10 Takten.
- **Ausführung der Monitore:** Die Kosten für die Abfrage des Monitors hängen von dessen Implementierung ab, insofern lassen sich hier keine allgemeinen Werte angeben. Während einfache Monitore nur einen bereits existierenden Wert zurückgeben, müssen andere erst noch eine Auswertung bestimmter Systemparameter durchführen. Tabelle 7.5 führt die Kosten der hier verwendeten Monitore auf. Man sieht, dass die Kosten für den Migrationsmonitor konstant sind, da dessen Rückgabewert innerhalb des Migrationsmoduls gespeichert ist und nur von dessen Aktoren verändert wird. Änliches gilt auch für den Watchdog-Monitor. Die Kosten der anderen Monitore hingegen können geringfügig variieren.

Klassifizierung Die Kosten der Klassifizierungsphase hängen im wesentlichen von der Anzahl der zu untersuchenden Aktoren ab. Die Laufzeit für die Untersuchung eines einzelnen Aktors beträgt dabei im Schnitt 156 Takte. Unabhängig von den Aktoren entstehen zudem Fixkosten von 139 Takten.

Tabelle 7.6: Kosten für die Ausführung von Aktoren

Aktor	Kosten (Takte)
IncFreqAktor	286 / 128 (Operation nicht verfügbar)
DecFreqAktor	286 / 128 (Operation nicht verfügbar)
Emigrationsaktor	$1208 + n \times 1038$ (siehe 7.6)
Immigrationsaktor	813

Reaktion Die Kosten der Reaktion werden maßgeblich von dem ausgewählten Aktor bestimmt. Tabelle 7.6 gibt die Laufzeiten der hier verwendeten Aktoren an. Zu diesen Zeiten kommen noch fixe Kosten von DCERT in Höhe von 229 Takten.

7.7.2 Fazit

Ziel beim Entwurf von DCERT war ein problemunabhängiger Algorithmus mit geringen Ausführungskosten. Die Problemunabhängigkeit erreicht DCERT durch die Verwendung von Statusnachrichten, deren Bedeutung durch den Einsatzbereich definiert wird. Die Ausführungkosten, die durch den DCERT-Algorithmus an sich entstehen, betragen in dem hier vorgestellten Szenario mit fünf Monitoren und vier Aktoren nur knapp 1.200 Takte. Hinzu kommen noch die Ausführungskosten für die Monitore sowie die gewählte Reaktion.

Die Ausführungskosten von DCERT hängen also nur von der Anzahl der Monitore und Aktoren ab. Insofern ist DCERT im Vorteil gegenüber automatischen Planern. Diese sind zwar in der Lage, eine komplette Reaktionenfolge herzuleiten, während DCERT in einem Durchlauf nur eine Reaktion herleitet. Ein Planer muss allerdings alle möglichen Zustandsübergänge betrachten, die durch Aktionen entstehen können. Dadurch steigt der Planungsaufwand exponentiell mit der Größe des Suchraums.

Da der DCERT-Algorithmus nicht fortlaufend ausgeführt werden muss, sondern nur beim Auftreten eines Missstandes, entsteht nur ein geringer Overhead für das Basissystem. Trotzdem ist es nötig, DCERT als isolierten Helper Thread auszuführen. So wird zum einen vermieden, dass seine Ausführung von einem umgebenden Programm beeinflusst wird, zum anderen wird hier wiederum der Einfluss auf das Zeitverhalten parallel arbeitender Echtzeitanwendungen verhindert.

8 Zusammenfassung

Dieses Kapitel fasst die Konzepte, Umsetzungen und Ergebnisse dieser Arbeit zusammen. Abschließend gibt es einen Ausblick auf mögliche zukünftige Arbeiten bei der Integration von Konzepten des Organic Computing in eingebetteten Echtzeitsystemen.

8.1 Zusammenfassung

Die Komplexität eingebetteter Systeme wächst stetig. Durch die Vernetzung einzelner Steuereinheiten entstehen verteilte Systeme, deren Verwaltung und Wartung immer schwieriger wird. Erschwert wird dies durch den Umstand, dass viele der Steuereinheiten unter Echtzeitbedingungen arbeiten müssen. Das Zeitverhalten darauf laufender Anwendungen darf auf keinen Fall negativ beeinflusst werden.

Die Paradigma des *Autonomic Computing* und *Organic Computing* können helfen, sowohl einzelne eingebettete Steuereinheiten als auch komplexe Netze solcher ECUs beherrschbar zu halten und deren Leistungsfähigkeit besser auszunutzen. Die Bereitstellung der Selbst-X-Fähigkeiten Selbstkonfiguration, Selbstheilung, Selbstoptimierung und Selbstschutz reduziert den Administrationsaufwand für solche Systeme, erleichtert deren Entwicklung und erhöht deren Ausfallsicherheit. Dabei dürfen aber nie die Echtzeitanforderungen an die Steuereinheiten aus den Augen verloren werden. Die Entwicklung mehrfädiger Prozessorarchitekturen für eingebettete Systeme liefert hier eine wichtige Unterstützung.

Die Integration von Selbst-X-Fähigkeiten für eingebettete Steuereinheiten muss beim Entwurf der Knotensoftware bereits auf unterster Ebene vorbereitet werden. Ein Echtzeitbetriebssystem, das auf solchen Steuereinheiten verwendet wird, muss dazu entsprechend erweitert werden. Grundlage aller Selbst-X-Techniken ist die Überwachung relevanter Systemparameter. Das Betriebssystem muss dazu umfassende Informationen über das aktuelle Systemverhalten zur Verfügung stellen können und entsprechende Eingriffsmöglichkeiten bieten, mit denen das Systemverhalten auch gezielt beeinflusst werden kann. Durch geeignete Sicherheitstechniken muss das Betriebssystem aber auch sicherstellen, dass durch die Eingriffe keine Funktionsstörungen ausgelöst werden. Insbesondere dürfen die

Selbst-X-Techniken laufende Echtzeitanwendungen nicht in ihrer Funktionsfähigkeit und ihrem Zeitverhalten beeinträchtigen. Für die Adaptionsfähigkeit von verteilten Systemen kann es zudem nötig sein, einzelne Anwendungen zwischen Steuereinheiten zu verschieben. Das Betriebssystem muss deshalb über geeignete Techniken zur Codemigration verfügen. Sowohl der Prozessor als auch das Betriebssystem müssen es erlauben, die Selbst-X-Techniken in zeitlicher Isolation von Echtzeitanwendungen auszuführen, um deren Zeitverhalten nicht zu beeinflussen.

Die Architektur von CAROS greift diese Anforderungen auf. Ein *Thread Management* stellt die Mittel zur Verfügung, die für die Ausführung von Echtzeitanwendungen nötig sind. Zudem erlaubt es den Einsatz von Helper Threads, die die Anwendung unterstützen können, aber zeitlich von ihr isoliert sind. Dazu nutzt es den bereits in Hardware vorhandenen Scheduler des simultan mehrfädigen CarCore-Prozessors. Eine dynamische Speicherverwaltung erlaubt Anwendungen eine flexible Nutzung des Arbeitsspeichers. Durch die Speicherallokation auf zwei Ebenen wird eine mögliche Beeinflussung von Anwendungen untereinander minimiert. Die Nutzung vorhersagbarer Allokationstechniken erlaubt den Einsatz des Haldenspeicher auch innerhalb von Echtzeitanwendungen. Die dynamische Speicherverwaltung bildet eine wichtige Basis für Ressourcenverwaltung und den für die Codemigration nötigen Runtime Linker. Aufgabe der Ressourcenverwaltung ist es, den Zugriff auf über Prozessor und Speicher hinausgehende Hardwareressourcen zu kontrollieren. Zudem verwaltet sie die Gerätetreiber, die für diese Zugriffe benötigt werden. Diese Treiber sind in CAROS dynamisch gebunden und können zur Laufzeit ausgetauscht werden. Dadurch ist es jederzeit möglich, Aktualisierungen an den Treibern vorzunehmen. Der Runtime Linker stellt die technische Voraussetzung zur Migration von Anwendungen bereit. Er kann zudem genutzt werden, um Programmbibliotheken zu laden, auf die von mehreren Anwendungen zugegriffen wird. Alle Zugriffe auf den CAROS-Kern werden von einem Security Manager überwacht. Dieser prüft die Rechte des zugreifenden Threads und verweigert den Zugriff, falls der Thread nicht über ausreichende Rechte verfügt. Der CAROS-Kern stellt damit die Basis dar, um innerhalb eines eingebetteten Echtzeitsystems die Selbst-X-Fähigkeiten des Autonomic/Organic Computing in Form eines Autonomic Managements zu implementieren.

Um den Echtzeitbetrieb der Steuereinheit zu unterstützen, ist das Autonomic Management in zwei Schichten geteilt. Die untere Schicht besteht aus kompakten Modulmanagern, die an konkrete Hard- oder Software-Module gebunden sind. Sie verarbeiten nur einen kleinen Parametersatz und können gegebenenfalls auch in Echtzeitanwendungen integriert werden. Sollte ihr Parametersatz Missstände aufzeigen, so versuchen sie zunächst selbstständig eine Beseitigung innerhalb ihres Moduls. Nur wenn dies nicht nötig ist, melden sie das Problem in abstrahierter Form an die obere Schicht, den Knotenmanager weiter. Dieser nimmt seine Ar-

beit nur auf, wenn er von einem Missstand Kenntnis erhält. Dann sammelt er die abstrahierten Zustandsinformationen aller verfügbaren Modulmanager. Mit Hilfe des generischen Algorithmus' DCERT leitet er daraus eine Reaktion her, die er mithilfe eines Modulmanagers ausführt. Der DCERT-Algorithmus selbst ist so entworfen, dass er die Ressourcenbeschränktheit eingebetteter Systeme berücksichtigt. Die abstrakten Zustandsinformationen der Modulmanager werden als 1-Bit-Werte innerhalb von Ganzzahlvariablen dargestellt. Dies ermöglicht eine effiziente Verarbeitung des Systemzustands unter Nutzung der auf den meisten Prozessoren verfügbaren bitweisen Operationen.

Mehrere Modulmanager verdeutlichen beispielhaft die Arbeitsweise der zweischichtigen Managementarchitektur. Ein *Modulmanager zur Schedulingadaption* kann helfen, den Energieverbrauch von periodischen iterativen weichen Echtzeitanwendungen zu senken. Mithilfe des *Autocorrelation Clustering* sagt er den Rechenaufwand einzelner Iterationen voraus und passt darauf basierend die Taktfrequenz und Schedulingparameter des Anwendungsthreads an. Für den Einsatz mit DCERT verzichtet er auf die Frequenzanpassung, welche DCERT nach Bedarf selbst vornimmt. Grundlage für diese Entscheidungen sind die Informationen, die DCERT von einem *Schedulingmonitor* erhält. Dieser überwacht das Zeitverhalten des Hardware-Schedulers im simultan mehrfädigen CarCore-Prozessor. Ein *Software-Watchdog-Dienst* kann dazu genutzt werden, den korrekten Ablauf einer Anwendung zu überwachen. Im Gegensatz zu einem Hardware-Watchdog, der bei einem Fehler gewöhnlich den gesamten Prozessor zurücksetzt, erlaubt der Software-Watchdog eine differenziertere Reaktion. Sollte dabei die von der Anwendung vorgegebene Fehlerroutine nicht dauerhaft zu einer Lösung des Problems führen, kann der Software-Watchdog die Fehlerinformationen zur genaueren Bearbeitung an DCERT weiterleiten. Ein *Migrationsmanager* kümmert sich um das Verschieben von Anwendungen zwischen Knoten. Die Entscheidung, ob eine Anwendung verschoben oder empfangen wird, muss DCERT treffen. Der Migrationsmanager aber trifft im Emigrationsfall selbstständig die Entscheidung, auf welchen Knoten eine Anwendung verschoben wird. Auch prüft er eigenständig die Praktikabilität möglicher Immigrationen.

Die Evaluierungen von DCERT und der Modulmanager unterstreichen klar die Notwendigkeit von Helper Threads. Gerade aufwändige Modulmanager wie etwa der Migrationsmanager müssen isoliert von möglichen Echtzeitanwendungen ausgeführt werden können. Aber auch DCERT selbst muss von laufenden Anwendungen entkoppelt sein, um bei Missständen möglichst schnell reagieren zu können.

Diese Ergebnisse zeigen, wie den Einschränkungen eingebetteter Echtzeitsysteme zu begegnen ist, um dort Selbst-X-Techniken zu integrieren. Aus dieser Integration kann man schon auf Ebene einzelner Steuereinheiten profitieren.

8.2 Ausblick

Diese Arbeit konzentriert sich überwiegend auf die Betrachtung einer einzelnen Steuereinheit. Problematiken, die in verteilten Systemen entstehen, werden nur am Rande betrachtet. Aber auch im Bereich organischer Middlewaresysteme existieren schon beachtliche Arbeiten [59, 40, 41]. Eine Erweiterung des CAROS/DCERT-Systems in diese Richtung oder eine Integration mit diesen Systemen würde den Ansatz des Organic Computing in eingebetteten Echtzeitsystemen vervollständigen. Dies erscheint gerade unter dem Gesichtspunkt wichtig, dass die Automatisierung und Vernetzung auch des alltäglichen Lebens weiter zunehmen wird.

Aber auch im Hinblick auf die Modulmanager bestehen durchaus Möglichkeiten zu einer Fortführung dieser Arbeit. Eingebettete Systeme verfügen auch schon heute über eine Vielzahl von Peripheriegeräten, die auch als Sensoren und Aktuatoren genutzt werden können. Gerade verschiedene Kommunikationstechniken wie Wireless LAN (IEEE 802.11) oder Wireless PAN (IEEE 802.15: Bluetooth, Zigbee, etc.) eröffnen hier neue Möglichkeiten für Modulmanager. Hier sind Techniken denkbar, die selbstständig den jeweils besten Kommunikationsweg wählen. Kriterien können hier unter anderem Energieeffizienz oder auch Zuverlässigkeit sein. Bei Übertragungsproblemen könnte dann automatisch eine andere Technik gewählt werden, ohne dass eine darauf aufbauende Anwendung davon etwas mitbekäme. Ähnliche Prinzipien können auch für die Interaktion mit einem Benutzer bei der Wahl von Ein- und Ausgabegeräten angewendet werden.

Literaturverzeichnis

[1] Agarwal, A.: *Performance Tradeoffs in Multithreaded Processors.* IEEE Transactions on Parallel and Distributed Systems, 3(5):525–539, 1992.

[2] *AUTOSAR AUTomotive Open System ARchitecture.* http://www.autosar.org/.

[3] AUTOSAR GbR: *AUTOSAR Technical Overview*, 2.2.2 Auflage, August 2008.

[4] Baker, Theodore P.: *A Stack-Based Resource Allocation Policy for Realtime Processes.* In: *IEEE Real-Time Systems Symposium*, Seiten 191–200, 1990.

[5] Berberidis, Christos, Walid G. Aref, Mikhail Atallah, Ioannis Vlahavas und Ahmed K. Elmagarmid: *Multiple and Partial Periodicity Mining in Time Series Databases.* In: *In Proc. of the 15th Euro. Conf. on Artificial Intelligence*, 2002.

[6] Brinkschulte, Uwe, Christian Krakowski, J. Riemschneider, Jochen Kreuzinger, Matthias Pfeffer und Theo Ungerer: *A microkernel architecture for a highly scalable real-time middleware.* In: *RTAS 2000, 6th IEEE Real-time Technology and Application Symposium, Work in Progress session*, Mai 2000.

[7] Brinkschulte, Uwe, Mathias Pacher, Florentin Picioroaga und Stefan Gaa: *Evaluation of the Komodo Microcontroller and the OSA+ Middleware Using an Autonomous Guided Vehicle.* In: *ISORC*, Seiten 550–557. IEEE Computer Society, 2006.

[8] Brinkschulte, Uwe, Etienne Schneider und Florentin Picioroaga: *Dynamic Real-time Reconfiguration in Distributed Systems: Timing Issues and Solutions.* In: *ISORC*, Seiten 174–181. IEEE Computer Society, 2005.

[9] Campbell, Roy, Garry Johnston und Vincent Russo: *Choices (class hierarchical open interface for custom embedded systems).* SIGOPS Oper. Syst. Rev., 21(3):9–17, 1987.

[10] Chappell, Robert S., Jared Stark, Sangwook P. Kim, Steven K.

Reinhardt und Yale N. Patt: *Simultaneous Subordinate Microthreading (SSMT).* In: *ISCA*, Seiten 186–195, 1999.

[11] Collins, Jamison D., Hong Wang, Dean M. Tullsen, Christopher J. Hughes, Yong-Fong Lee, Daniel M. Lavery und John Paul Shen: *Speculative precomputation: long-range prefetching of delinquent loads.* In: *ISCA*, Seiten 14–25, 2001.

[12] David, Francis M. und Roy H. Campbell: *Building a Self-Healing Operating System.* In: *DASC '07: Proceedings of the Third IEEE International Symposium on Dependable, Autonomic and Secure Computing*, Seiten 3–10, Washington, DC, USA, 2007. IEEE Computer Society.

[13] Ghallab, Malik, Dana Nau und Paolo Traverso: *Automated Planning: Theory and Practice.* Morgan Kaufman, San Francisco, CA, 2004.

[14] *HIS Herstellerinitiative Software.* http://www.automotive-his.de.

[15] Holland, John H.: *Processing and processors for schemata.* In: Jacks, E. L. (Herausgeber): *Associative Information Processing*, Seiten 127–146. New York: American Elsevier, 1971.

[16] Horn, Paul: *Autonomic Computing: IBM's Perspective on the State of Information Technology.* IBM Manifesto, IBM Corporation, Oktober 2001.

[17] Infineon Technologies AG: *TriCore 1 User's Manual*, Januar 2008. V1.3.8.

[18] Intel Corporation: *Intel PXA270 Processor (Data Sheet)*, 2005.

[19] Keckler, Stephen W., Andrew Chang, Whay Sing Lee, Sandeep Chatterjee und William J. Dally: *Concurrent Event Handling through Multithreading.* IEEE Trans. Computers, 48(9):903–916, 1999.

[20] Kephart, Jeffrey O. und David M. Chess: *The Vision of Autonomic Computing.* IEEE Computer, 36(1):41–50, 2003.

[21] Kluge, Florian, Jörg Mische, Sascha Uhrig und Theo Ungerer: *An Operating System Architecture for Organic Computing in Embedded Real-Time Systems.* In: *Proceedings of the 5th International Conference on Autonomic and Trusted Computing (ATC 2008)*, Seiten 343–357, Oslo, Norway, Juni 2008. Springer.

[22] Kluge, Florian, Jörg Mische, Sascha Uhrig und Theo Ungerer: *Building Adaptive Embeddd Systems by Monitoring and Dynamic Loading of Application Modules.* In: *Workshop on Adaptive and Reconfigurable Embedded Systems (APRES'08)*, Seiten 23–26, St. Louis, MO, USA, April 2008.

[23] Kluge, Florian, Jörg Mische, Sascha Uhrig, Theo Ungerer und

Rafael Zalman: *Use of Helper Threads for OS Support in the Multithreaded Embedded TriCore 2 Processor*. In: Lu, Chenyang (Herausgeber): *Proceedings Work-In-Progress-Session of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium*, Seiten 25–27, April 2007.

[24] Kluge, Florian, Sascha Uhrig, Jörg Mische und Theo Ungerer: *A Two-Layered Management Architecture for Building Adaptive Real-time Systems*. In: *Proceedings of The 6th IFIP Workshop on Software Technologies for Future Embedded & Ubiquitous Systems (SEUS 2008)*, Seiten 126–137, Capri Island, Italy, 2008. Springer.

[25] Kluge, Florian, Sascha Uhrig, Jörg Mische, Benjamin Satzger und Theo Ungerer: *Dynamic Workload Prediction for Soft Real-Time Applications*. In: *Accepted for publication at the 7th IEEE International Conferences on Embedded Software and Systems (ICESS-10), Bradford, United Kingdom, June 29 - July 1, 2010, Proceedings*, Bradford, UK, Juni 2010.

[26] Kluge, Florian, Sascha Uhrig, Jörg Mische, Benjamin Satzger und Theo Ungerer: *Optimisation of Energy Consumption of Soft Real-Time Applications by Workload Prediction*. In: *First IEEE Workshop on Self-Organizing Real-Time Systems (SORT 2010), Carmona, Spain, May 4, 2010, Proceedings*, Seiten 63–72, Carmona, Spain, Mai 2010.

[27] Kluge, Florian, Chenglong Yu, Jörg Mische, Sascha Uhrig und Theo Ungerer: *Implementing AUTOSAR Scheduling and Resource Management on an Embedded SMT Processor*. In: *12th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2009)*, Seiten 33–41, Nice, France, April 2009.

[28] Kreuzinger, Jochen, Alexander Schulz, Matthias Pfeffer, Theo Ungerer, Uwe Brinkschulte und Christian Krakowski: *Real-time Scheduling on Multithreaded Processors*. In: *7th Int. Conference on Real-Time Computing Systems and Applications*, Seiten 155–159, Dezember 2000.

[29] Laudon, James, Anoop Gupta und Mark Horowitz: *Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations*. In: *In Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, Seiten 308–318, 1994.

[30] Lea, Doug: *A Memory Allocator*. unix/mail, Dezember 1996.

[31] Liu, C. L. und James W. Layland: *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. Journal of the ACM, 20(1):46–61, Januar 1973.

[32] Luk, Chi-Keung: *Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors*. In: *ISCA*, Seiten

40–51, 2001.

[33] Manacher, Glenn K.: *Production and Stabilization of Real-Time Task Schedules.* J. ACM, 14(3):439–465, 1967.

[34] Marcuello, Pedro, Antonio González und Jordi Tubella: *Speculative Multithreaded Processors.* In: *International Conference on Supercomputing*, Seiten 77–84, 1998.

[35] Masmano, Miguel, Ismael Ripoll, Alfons Crespo und Jorge Real: *TLSF: A New Dynamic Memory Allocator for Real-Time Systems.* In: *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)*, Seiten 79–86, Washington, DC, USA, 2004. IEEE Computer Society.

[36] Mische, Jörg, Irakli Guliashvili, Sascha Uhrig und Theo Ungerer: *How to Enhance a Superscalar Processor to Provide Hard Real-Time Capable In-Order SMT.* In: Christian Müller-Schloer, Wolfgang Karl, Sami Yehia (Herausgeber): *23rd International Conference on Architecture of Computing Systems (ARCS 2010), Hannover, Germany, February 22-25, 2010, Proceedings*, Band 5974 der Reihe *Lecture Notes in Computer Science*, Seiten 2–14, Hannover, Germany, Februar 2010. Springer.

[37] Mische, Jörg, Sascha Uhrig, Florian Kluge und Theo Ungerer: *Exploiting Spare Resources of In-order SMT Processors Executing Hard Real-time Threads.* In: *IEEE International Conference on Computer Design 2008 (ICCD 08)*, Seiten 371–376, Lake Tahoe, CA, USA, Oktober 2008.

[38] Mische, Jörg, Sascha Uhrig, Florian Kluge und Theo Ungerer: *IPC Control for Multiple Real-Time Threads on an In-order SMT Processor.* In: *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC 2009)*, Seiten 125–139, Paphos, Cyprus, Januar 2009.

[39] Müller-Schloer, C.: *Organic computing: on the feasibility of controlled emergence.* In: *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, Seiten 2–5, New York, NY, USA, 2004. ACM Press.

[40] Nickschas, Manuel und Uwe Brinkschulte: *Using Multi-Agent Principles for Implementing an Organic Real-Time Middleware.* In: *Proc. 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, Seiten 189–195, Santorini, Greece, 2007.

[41] Nickschas, Manuel und Uwe Brinkschulte: *Guiding Organic Management in a Service-Oriented Real-Time Middleware Architecture.* In: *Procee-*

dings of The 6th IFIP Workshop on Software Technologies for Future Embedded & Ubiquitous Systems (SEUS 2008), Capri Island, Italy, 2008. Springer.

[42] Nyquist, H.: *Certain topics in telegraph transmission theory.* Transactions of the AIEE, 47:617–644, 1928.

[43] *OSEK VDX Portal.* http://www.osek-vdx.org.

[44] Pfeffer, Matthias und Theo Ungerer: *Dynamic Real-Time Reconfiguration on a Multithreaded Java-Microcontroller.* In: *International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004), Vienna, Austria*, Seiten 86–92, Mai 2004.

[45] Pfeffer, Matthias, Theo Ungerer, Stephan Fuhrmann, Jochen Kreuzinger und Uwe Brinkschulte: *Real-Time Garbage Collection for a Multithreaded Java Microcontroller.* Real-Time Systems, 26(1):89–106, 2004.

[46] Picioroaga, Florentin und Uwe Brinkschulte: *Flexible QoS management and real-time in OSA+ middleware.* In: Arabnia, Hamid R. (Herausgeber): *PDPTA*, Seiten 984–990. CSREA Press, 2006.

[47] *IEEE Std 1003.1, 2004 Edition. The Open Group Base Specifications Issue 6*, 2004.

[48] *QNX Software Systems.* http://www.qnx.com/.

[49] Rajkumar, Ragunathan, Lui Sha und John P. Lehoczky: *Real-Time Synchronization Protocols for Multiprocessors.* In: *IEEE Real-Time Systems Symposium*, Seiten 259–269. IEEE Computer Society, 1988.

[50] Richter, Urban, Moez Mnif, Jürgen Branke, Christian Müller-Schloer und Hartmut Schmeck: *Towards a Generic Observer/Controller Architecture for Organic Computing.* In: *Informatik 2006 - Informatik für Menschen, Band 1, Beiträge der 36. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 2.-6. Oktober 2006 in Dresden*, Seiten 112–119, 2006.

[51] Russell, Stuart und Peter Norvig: *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[52] Satzger, Benjamin, Andreas Pietzowski, Wolfgang Trumler und Theo Ungerer: *Using Automated Planning for Trusted Self-organising Organic Computing Systems.* In: *Proceedings of the 5th International Conference on Autonomic and Trusted Computing (ATC 2008)*, Seiten 60–72, Oslo, Norway, Juni 2008. Springer.

[53] Schneider, Etienne, Florentin Picioroaga und Uwe Brinkschul-

TE: *Dynamic reconfiguration through OSA+, a real-time middleware.* In: CURRY, EDWARD, DOUG LEA und FABIO KON (Herausgeber): *Doctoral Symposium on Middleware*, Seiten 319–323. ACM, 2004.

[54] SCHÖLER, THORSTEN und CHRISTIAN MÜLLER-SCHLOER: *First steps towards organic computing systems: monitoring an adaptive protocol stack with a fuzzy classifier system.* In: *CF '05: Proceedings of the 2nd conference on Computing frontiers*, Seiten 10–20, New York, NY, USA, 2005. ACM Press.

[55] SCHUSTER, ARTHUR: *On the Investigation of Hidden Periodicities With Application to a Supposed 26 Day Period of Meteorological Phenomena.* Terrestrial Magnetism, 3:13–41, 1898.

[56] SHA, LUI, RAGUNATHAN RAJKUMAR und JOHN P. LEHOCZKY: *Priority Inheritance Protocols: An Approach to Real-Time Synchronization.* IEEE Trans. Comput., 39(9):1175–1185, 1990.

[57] SHANNON, CLAUDE ELWOOD: *Communication in the Presence of Noise.* Proceedings of the IRE, 37(1):10–21, 1949.

[58] *tresos ® ECU.* http://www.tresos.de/.

[59] TRUMLER, WOLFGANG, MARKUS HELBIG, ANDREAS PIETZOWSKI, BENJAMIN SATZGER und THEO UNGERER: *Self-Configuration and Self-Healing in AUTOSAR.* In: *14th Asia Pacific Automotive Engineering Conference*, Hollywood, California, USA, August 2007. SAE International.

[60] TULLSEN, DEAN M., SUSAN J. EGGERS und HENRY M. LEVY: *Simultaneous multithreading: maximizing on-chip parallelism.* In: *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, Seiten 533–544, New York, NY, USA, 1998. ACM.

[61] UHRIG, SASCHA, STEFAN MAIER und THEO UNGERER: *Toward a Processor Core for Real-time Capable Autonomic Systems.* In: *Proceedings of the 5th IEEE International Symposium on Signal Processing and Information Technology*, Seiten 19–22, Dezember 2005.

[62] UHRIG, SASCHA und JÖRG WIESE: *jamuth – An IP Processor Core for Embedded Java Real-Time Systems.* In: *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, 2007.

[63] VENIS, MARC: *Testing of NiCd and NiMH Batteries.* http://www.vencon.com/index.php?page=support_art1, besucht am 9. Juni 2010, Januar 2008.

[64] VLACHOS, MICHAIL, PHILIP S. YU und VITTORIO CASTELLI: *On Periodicity Detection and Structural Periodic Similarity.* In: *SDM*, 2005.

[65] Wind River Systems Inc.: *Wind River General Purpose Platform, VxWorks Edition 3.6.*

[66] Zilles, Craig B., Joel S. Emer und Gurindar S. Sohi: *The Use of Multithreading for Exception Handling.* In: *MICRO*, Seiten 219–229, 1999.

[67] Zilles, Craig B. und Gurindar S. Sohi: *Execution-based prediction using speculative slices.* In: *ISCA*, Seiten 2–13, 2001.

Abbildungsverzeichnis

Tabellenverzeichnis

A Methoden zur Periodizitätsbestimmung

Die Bestimmung von Periodizitäten in Datenreihen ist eine weit verbreitete Problemstellung. Sie findet in vielen Bereichen Anwendung. Viele natürliche Vorgänge weisen eine Regelmäßigkeit in ihrem Ablauf auf. In den Naturwissenschaften können auf diese Weise etwa Gezeitenmuster oder Temperaturänderungen analysiert werden. Ebenso finden sich im menschlichen Körper viele regelmäßige Vorgänge, wie etwa der Herzschlag. In der Medizin können Techniken zur Periodizitätsbestimmung genutzt werden, um Unregelmäßigkeiten in diesen Vorgängen zu entdecken und so Risiken möglichst frühzeitig zu erkennen. Auf ähnliche Weise können auch in technischen Prozessen Fehler in Maschinen erkannt werden. Damit sind die tatsächlichen Einsatzgebiete der Periodizitätsbestimmung aber bei weitem nicht erschöpft.

Die folgenden Abschnitte erläutern zwei Techniken, mit deren Hilfe Periodizitäten in Datenreihen maschinell bestimmt werden können. Beide Techniken hängen mit der Diskreten Fourier-Transformation zusammen, die in Abschnitt A.1 beschrieben wird. Abschnitt A.2 zeigt, wie man mithilfe eines Periodogramms die Periodizität einer Datenreihe bestimmen kann. Abschnitt A.3 führt die zyklische Autokorrelation als zweite Methode ein. Ein Vergleich der Methoden erfolgt in Abschnitt A.4.

A.1 Diskrete Fourier-Transformation

Bei der *Diskreten Fourier-Transformation (DFT)* handelt es sich um die Fourier-Transformation eines zeitdiskreten periodischen Signals. Das Ergebnis einer solchen Transformation ist das Frequenzspektrum des Signals. Dieses wird als Überlagerung eines Grundpegels, einer Grundschwingung sowie von Oberschwingungen beschrieben. In diesem Frequenzspektrum ist eine Analyse und auch gezielte Manipulation einzelner Frequenzen möglich. Da es sich bei der DFT um eine ein-eindeutige Abbildung handelt, ist eine Rücktransformation in den Zeitraum möglich.

Das Signal wird als Folge oder Vektor komplexer Zahlen $x_n \in \mathbb{C}, n = 0, 1, \ldots, N-1$ dargestellt. Die diskrete Fourier-Transformation ist folgendermaßen definiert:

$$\hat{x}_k = \sum_{n=0}^{N-1} x_n \mathrm{e}^{-2\pi \mathrm{i} \frac{kn}{N}}, \quad k = 0, 1, \ldots, N-1 \tag{A.1}$$

Die Rücktransformation aus dem Frequenzraum in den Zeitraum erfolgt mit der inversen diskreten Fourier-Transformation (IDFT):

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} \hat{x}_k \mathrm{e}^{2\pi \mathrm{i} \frac{kn}{N}}, \quad n = 0, 1, \ldots, N-1 \tag{A.2}$$

Die diskrete Fouriertransformation kann auch auf einen Vektor reeller Zahlen angewendet werden. In diesem Fall erhält man nur $\frac{N}{2}$ Frequenzen, die in Superposition wiederum das Originalsignal ergeben.

Alternativ kann die diskrete Fourier-Transformation auch als lineare Abbildung $\hat{x} = Fx$ dargestellt werden. Die Einträge des Vektors x sind dabei die Signalwerte x_n zu den Abtastzeitpunkten. $F \in \mathbb{C}^{N \times N}$ steht für die Fouriermatrix mit Einträgen F_{jk}:

$$F_{jk} = \mathrm{e}^{2\pi \mathrm{i} \frac{jk}{N}} \tag{A.3}$$

Die Fourier-Matrix F ist invertierbar, und für ihre inverse Matrix F^{-1} gilt:

$$F_{jk}^{-1} = \frac{1}{N} \mathrm{e}^{-2\pi \mathrm{i} \frac{jk}{N}} \tag{A.4}$$

Zur Berechnung der diskreten Fourier-Transformation wird meist auf die *Fast-Fourier-Transformation* zurückgegriffen. Durch die Ausführung der Matrixmultiplikation in einer bestimmten Reihenfolge und das Zurückgreifen auf bereits berechnete Zwischenergebnisse kann die Fast-Fourier-Transformation mit einer Laufzeitkomplexität von $O(N \log N)$ ausgeführt werden. Dabei ist allerdings die Länge N des Eingabevektors x auf Zweierpotenzen beschränkt ($N = 2^k$).

Aus dem Nyquist-Shannon-Abtasttheorem [42, 57] ergibt sich eine Beschränkung für die Anwendbarkeit der diskreten Fourier-Transformation. Deren Ergebnisse sind nur dann korrekt, wenn die Abtastfrequenz mindestens das doppelte der maximal auftretenden Signalfrequenz beträgt.

A.2 Periodogramm

Auf Basis der diskreten Fourier-Transformation ist es nun möglich, ein Periodogramm [55, 64] zu berechnen. Dieses dient der Schätzung der Spektraldichte eines

Signals. Die Berechnung erfolgt nach Formel A.5:

$$P(\hat{x}_n) = \|\hat{x}_n\|^2, \quad n = 0.. \left\lceil \frac{N-1}{2} \right\rceil \tag{A.5}$$

Die Maxima innerhalb des Periodogramms stehen für dominante Frequenzen in dem Signal. Die Periodenlänge P errechnet sich über $P = \frac{1}{f}$. Dabei ist allerdings zu beachten, dass der Koeffizient $\hat{x}_n$ Perioden der Länge $\left[\frac{N}{n} \ldots \frac{N}{n-1}\right)$ beinhaltet. Dies erschwert die Analyse des Periodogramms.

Problematisch kann sich außerdem der *Leck-Effekt* äußern. Durch das Zeitfenster bei der Beobachtung wird das Eingangssignal abgeschnitten. Die diskrete Fourier-Transformation erfordert, dass das Eingangssignal periodisch fortsetzbar ist. Wenn dies nicht der Fall ist, so enthält das Eingangssignal Frequenzen, die nicht durch die diskrete Fourier-Transformation berechnet werden. Stattdessen nähert die Transformation diese Frequenzen an und verteilt deren Energie auf benachbarte Frequenzen.

A.3 Autokorrelation

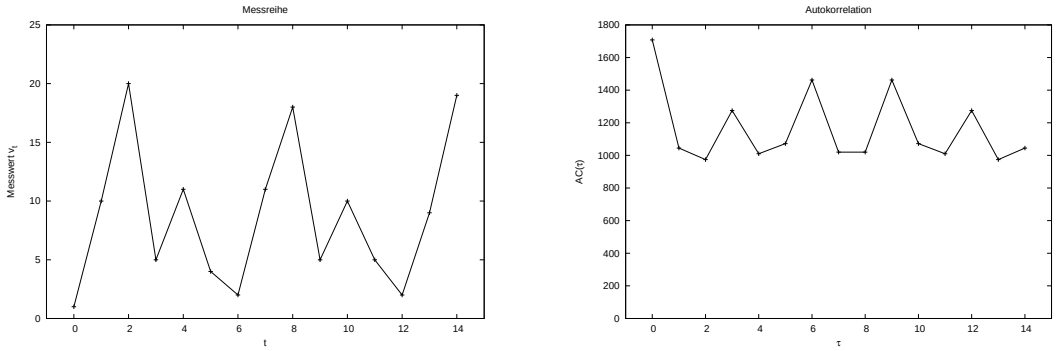
Eine weitere Methode, um Periodizitäten innerhalb einer Datenreihe zu bestimmen ist die *zyklische Autokorrelationsfunktion*. Sie ist ein Maß für die Korrelation zwischen der Datenreihe $x_0, \ldots, x_{N-1}$ und deren zyklischer Verschiebung um τ Einträge:

$$AC(\tau) = \sum_{n=0}^{N-1} x_k x_{(k+\tau) \bmod N}, \quad \tau \in \{0 \ldots N-1\} \tag{A.6}$$

Eine Periode $P = \tau_{\max}$ wird durch jene Verschiebung $\tau \neq 0$ angezeigt, an der die Autokorrelationsfunktion $AC(\tau)$ ihr Maximum annimmt:

$$AC(\tau_{\max}) = \max_{\tau \in \{1 \ldots N-1\}} AC(\tau) \tag{A.7}$$

Da die Autokorrelationsfunktion für $\tau \in 1 \ldots N-1$ achsensymmetrisch ist, müssen tatsächlich nur die Werte $\lceil 0 \ldots \frac{N-1}{2} \rceil$ für τ untersucht werden. Auch für die Periodizitätsbestimmung mit der Autokorrelation gilt die Einschränkung des Abtasttheorems. Eine Periode P kann nur dann erkannt werden, wenn die zugrunde liegende Datenreihe aus mindestens $2P$ Einträgen besteht.



(a) Messreihe (b) Autokorrelation

Abbildung A.1: Messreihe und Autokorrelation

Die zyklische Autokorrelation kann auch unter Zurhilfenahme der DFT berechnet werden [5, 64]:

$$AC = F^{-1} < \hat{x}, \hat{x^*} > \tag{A.8}$$

x^* stellt hier die komplex Konjugierte zu x dar, $< a, b >$ die komponentenweise Multiplikation zweier Vektoren a, b.

Damit ergibt sich prinzipiell eine effektivere Methode zur Berechnung der zyklischen Autokorrelation. Die Berechnung nach Gleichung A.6 hat eine Komplexität von $O(n^2)$, während die Berechnung mit Hilfe von FFT lediglich eine Komplexität von $O(n \log n)$ hat.

Abbildung A.1(a) zeigt eine beispielhafte Messreihe. In Abbildung A.1(b) sind die zugehörigen Werte der zyklischen Autokorrelationsfunktion aufgeführt. Relevant ist hier nur der Bereich $\tau \in [0 \dots 7]$, die restlichen Werte folgen aus der Achsensymmetrie. Wie zu erwarten, liegt das globale Maximum bei $\tau = 0$. Die Periode wird korrekt bei $\tau = 6$ erkannt. Daneben existiert aber ein weiteres lokales Maximum bei $\tau = 3$. Diese rührt von den Messwerten $1, 4, \dots$ her.

A.4 Vergleich

Dieser Abschnitt untersucht die Leistungsfähigkeit und insbesondere die tatsächliche Laufzeitkomplexität der beschriebenen Algorithmen. Im Einzelnen sind dies:

- Berechnung des Periodogramms via FFT (PER)
- Berechnung der Autokorrelation via FFT (AC-F)
- Berechnung der Autokorrelation nach Gleichung A.6 (AC)

Diese Berechnungen wurden jeweils auf Gleit- und Festkommaarithmetik implementiert und auf dem CarCore ausgeführt. Der Prozessor verfügt über keine Gleitkommaeinheit. Stattdessen werden die Gleitkommaoperationen mittels Bibliotheksfunktionen nachgebildet, weshalb die Gleitkommaimplementierungen entsprechend schlechter abschneiden. Abbildung A.2 zeigt die Laufzeiten für verschieden lange Datensätze.

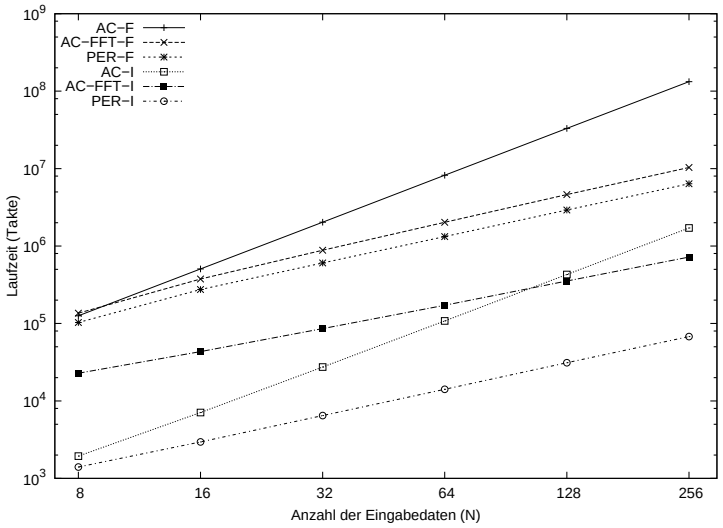


Abbildung A.2: Laufzeiten der Autokorrelationsberechnung

Bei den Gleitkommavarianten erzeugt die Berechnung der Autokorrelation nach Gleichung A.6 den höchsten Aufwand (Kurve AC-F). Die FFT-basierten Methoden unterscheiden sich in ihrer Gleitkommaimplementierung nur wenig (AC-FFT-F, PER-F). Bei den Implementierungen mit Festkommaarithmetik hingegen schneidet die FFT-basierte Autokorrelationsberechnung (AC-FFT-I) für Datensätze mit 64 oder weniger Einträgen schlechter ab als die Implementierung nach Gleichung A.6 (AC-I). Auch hier hat die Berechnung des Periodogramms (PER-I) die geringste Laufzeit. Damit spricht nicht nur die größere Flexibilität, die man durch die direkte Berechnung der Autokorrelation erhält für eine direkte Implementierung von Gleichung A.6. Bei den in Abschnitt 6.1 betrachteten Problemen wird eine eher niedrige Länge des Eingabefensters erwartet, so dass hier die Integer-Implementierung auch bezüglich der Laufzeit klar im Vorteil ist.

B Eigene Veröffentlichungen

Veröffentlichungen als Erstautor

2006

Kluge, Florian, Jörg Mische, Sascha Uhrig und Theo Ungerer: *CAR-SoC - Towards an Autonomic SoC Node.* In: *ACACES 2006 Poster Abstracts*, Seiten 37–40, L'Aquila, Italy, Juli 2006. Academia Press, Ghent (Belgium).

2007

Kluge, Florian, Jörg Mische, Sascha Uhrig, Theo Ungerer und Rafael Zalman: *Use of Helper Threads for OS Support in the Multithreaded Embedded TriCore 2 Processor.* In: Lu, Chenyang (Herausgeber): *Proceedings Work-In-Progress-Session of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium*, Seiten 25–27, April 2007.

Kluge, Florian, Jörg Mische, Stefan Metzlaff, Sascha Uhrig und Theo Ungerer: *Integration of Hard Real-Time and Organic Computing.* In: *ACACES 2007 Poster Abstracts*, Seiten 295–298, L'Aquila, Italy, Juli 2007. Academia Press, Ghent (Belgium).

2008

Kluge, Florian, Jörg Mische, Sascha Uhrig und Theo Ungerer: *Building Adaptive Embeddd Systems by Monitoring and Dynamic Loading of Application Modules.* In: *Workshop on Adaptive and Reconfigurable Embedded Systems (APRES'08)*, Seiten 23–26, St. Louis, MO, USA, April 2008.

Kluge, Florian und Julian Wolf: *Basic System-Level Software for a Single-Core MERASA Processor.* Technischer Bericht 2008-06, Institut für Informatik, Universität Augsburg, April 2008.

Kluge, Florian, Jörg Mische, Sascha Uhrig und Theo Ungerer: *An Operating System Architecture for Organic Computing in Embedded Real-Time*

Systems. In: *Proceedings of the 5th International Conference on Autonomic and Trusted Computing (ATC 2008)*, Seiten 343–357, Oslo, Norway, Juni 2008. Springer.

Kluge, Florian, Sascha Uhrig, Jörg Mische und Theo Ungerer: *A Two-Layered Management Architecture for Building Adaptive Real-time Systems*. In: *Proceedings of The 6th IFIP Workshop on Software Technologies for Future Embedded & Ubiquitous Systems (SEUS 2008)*, Seiten 126–137, Capri Island, Italy, 2008. Springer.

Kluge, Florian und Julian Wolf: *Refined System-Level Software for a Single-Core MERASA Processor*. Technischer Bericht 2008-15, Institut für Informatik, Universität Augsburg, Oktober 2008.

2009

Kluge, Florian, Chenglong Yu, Jörg Mische, Sascha Uhrig und Theo Ungerer: *Implementing AUTOSAR Scheduling and Resource Management on an Embedded SMT Processor*. In: *12th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2009)*, Seiten 33–41, Nice, France, April 2009.

Kluge, Florian und Julian Wolf: *System-Level Software for a Multi-Core MERASA Processor*. Technischer Bericht 2008-17, Institut für Informatik, Universität Augsburg, Oktober 2009.

2010

Kluge, Florian, Sascha Uhrig, Jörg Mische, Benjamin Satzger und Theo Ungerer: *Optimisation of Energy Consumption of Soft Real-Time Applications by Workload Prediction*. In: *First IEEE Workshop on Self-Organizing Real-Time Systems (SORT 2010), Carmona, Spain, May 4, 2010, Proceedings*, Seiten 63–72, Carmona, Spain, Mai 2010.

Kluge, Florian, Sascha Uhrig, Jörg Mische, Benjamin Satzger und Theo Ungerer: *Dynamic Workload Prediction for Soft Real-Time Applications*. In: *Accepted for publication at the 7th IEEE International Conferences on Embedded Software and Systems (ICESS-10), Bradford, United Kingdom, June 29 - July 1, 2010, Proceedings*, Bradford, UK, Juni 2010.

Weitere Veröffentlichungen

2008

Bagci, Faruk, Florian Kluge, Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler und Theo Ungerer: *Experiences with a Smart Office Project*. In: Yang, Laurence T. (Herausgeber): *Mobile Intelligence: Mobile Computing and Computational Intelligence*. Wiley-Interscience, Mai 2008.

Bagci, Faruk, Florian Kluge, Nader Bagherzadeh und Theo Ungerer: *LocSens - An Indoor Location Tracking System using Wireless Sensors*. In: *Proceedings of the 17th International Conference on Computer Communications and Networks, IEEE ICCCN 2008, St. Thomas, U.S. Virgin Islands, August 3-7, 2008*, Seiten 887–891, August 2008.

Mische, Jörg, Sascha Uhrig, Florian Kluge und Theo Ungerer: *Exploiting Spare Resources of In-order SMT Processors Executing Hard Real-time Threads*. In: *IEEE International Conference on Computer Design 2008 (ICCD 08)*, Seiten 371–376, Lake Tahoe, CA, USA, Oktober 2008.

2009

Mische, Jörg, Sascha Uhrig, Florian Kluge und Theo Ungerer: *IPC Control for Multiple Real-Time Threads on an In-order SMT Processor*. In: *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC 2009)*, Seiten 125–139, Paphos, Cyprus, Januar 2009.

Bagci, Faruk, Florian Kluge, Benjamin Satzger und Theo Ungerer: *Towards Indoor Location Estimation and Tracking with Wireless Sensors*. In: *IEEE International Symposium on Intelligent Signal Processing (WISP '09), Budapest, Hungary, August 26-28, 2009*, Seiten 235–240, August 2009.

Satzger, Benjamin, Florian Mutschelknaus, Faruk Bagci, Florian Kluge und Theo Ungerer: *Towards Trustworthy Self-optimization for Distributed Systems*. In: *Software Technologies for Embedded and Ubiquitous Systems, 7th IFIP WG 10.2 International Workshop, SEUS 2009, Newport Beach, CA, USA, November 16-18, 2009, Proceedings*, Seiten 58–68, November 2009.

Bagci, Faruk, Florian Kluge, Theo Ungerer und Nader Bagherzadeh: *Optimisations for LocSens – an indoor location tracking system using wireless sensors*. Intational Journal of Sensor Networks, 6(3/4):157–166, November 2009.

2010

Satzger, Benjamin, Faruk Bagci, Florian Kluge und Theo Ungerer: *Towards lightweight self-configuration in wireless sensor networks.* In: *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, Seiten 791–792, 2010.

Lebenslauf

Persönliche Daten

Name:	Florian Kluge
Geburtsdatum:	19. August 1979
Geburtsort:	Donauwörth

Ausbildung

1990-1999	Gymnasium Donauwörth Abschluss: Abitur
1999-2000	Grundwehrdienst / Zivildienst
2000-2005	Studium der Angewandten Informatik, Universität Augsburg Abschluss: Diplom-Informatiker
seit 2005	Wissenschaftlicher Angestellter am Lehrstuhl für Systemnahe Informatik und Kommunikationssysteme, Universität Augsburg