Thomas Gallenkamp:“Realisierung eines digitalen Echtzeitsimulators.“ Darmstädter Dissertation 1996 (Übersicht)

Die Simulation technischer Prozesse ist seit langem ein unentbehrliches Hilfsmittel bei der Entwicklung von Regelungs-, Steuerungs- und Schutzeinrichtungen. Die **digitale** Simulation hat sich deshalb weitestgehend durchgesetzt, weil sie im Vergleich zur Analog-Simulation bei drastisch sinkenden Kosten sehr flexibel und komfortabel einsetzbar ist. Die digitale Simulation auf leistungsfähigen PC´s bzw. Workstations gehört heute zu den Standard-Werkzeugen in der Entwurfsphase von Regelungs-, Steuerungs- und Schtzeinrichtungen. Dabei wird sowohl das Verhalten des technischen Prozeßes - in geeignet vereinfachter Form - als auch das der geplanten Regelungs-, Steuerungs- oder Schutzeinrichtung auf dem Digitalrechner nachgebildet. Eine Nachbildung in Echtzeit ist in der Entwurfsphase in der Regel nicht erforderlich, gleichwohl sollten ausufernde Simulationszeiten vermieden und die Gesamtkosten (Personal- und Simulator-Investitionskosten) minimiert werden.

Die entworfenen Regelungs-, Steuerungs- und Schutzeinrichtungen werden in der Regel digital, vorzugsweise mittels m-Rechnern realisiert. Die Überprüfung dieser Realisierung muß in **Echtzeit** geschehen. Es gibt zahlreiche Gründe, diese Überprüfung nicht gleich an der realen Anlage, sondern zuerst an einem Simulator, der das Verhalten der Anlage in Echtzeit nachbildet, vorzunehmen.

Herr Gallenkamp befasst sich primär mit der Realisierung der Hard- und vorallem der Software eines derartigen digitalen Echtzeitsimulators. Als Nebeneffekt ergibt sich, daß das damit geschaffene Gerät auch als leistungfähiger programmierbarer Regler eingesetzt werden kann.

Die Leistungsfähigkeit des Simulators soll modular erweiterbar sein, um sowohl für einfache als auch für umfangreiche und komplexe Prozesse eine angemessene Nachbildung zu ermöglichen. Daher wird, wie auch bei anderen Autoren, als Hardware des Simulators ein Parallelrechnersystem eingesetzt. Bisher fehlen jedoch grundlegende Ansätze zur Optimierung der auf dem Parallelrechner ablaufenden Simulationsprogramme, insbesondere die selbsttätige Aufteilung des Simulationsmodells in parallel rechenbare Teilmodelle. Herr Gallenkamp entwickelt daher einen Echtzeitsimulator, der hohe Simulationsleistung (Parallelrechner, optimierter Programmcode), automatische Parallelisierung und einfache Bedienung mittels grafischer Blockeingabe miteinander vereint.

Hardware

Grundsätzlich sollte für die einzelnen Rechner des Parallelrechners der zum Entwicklungszeitpunkt leistungsfähigste verfügbare Prozessortyp eingesetzt werden. Dies hätte im vorliegenden Fall die Neuentwicklung von Rechnerkarten mit z.B. Prozessoren des Typs Intel 860 oder Transputer bedeutet. Davon wurde abgesehen, weil zuvor am Institut für Stromrichtertechnik eine Rechnerkarte mit dem 32-bit Gleitkomma DSP TMS320C30 entwickelt wurde (Probst 1990), die im Institut derzeit als Standardlösung in zahlreichen Stromrichter- und Antriebsregelungen eingesetzt wird, so daß hierfür alle Entwicklungshilsmittel und ein breiter Erfahrungshintergrund vorlagen. Zwar stellt der TMS320C30 heute nicht die Spitze der Rechenleistung dar, ist aber trotzdem für die gestellte Aufgabe noch geeignet. Auf dieser vorhandenen Rechnerkarte (DIN 6U) sind am Primary Bus 128k schnelles statisches RAM (25ns, keine wait-states) sowie 128k EPROM vorhanden, der langsamere External Bus ist nur mit reduzierter Adress- und Datenbreite auf die Backplane herausgeführt. Im Aufbau von Herrn Gallenkamp werden drei dieser Rechnerkarten (im folgenden Rechenelemente genannt) eingesetzt, die über einen gemeinsamen Speicher untereinander und mit der PC-Schnittstelle Daten austauschen können (s.u.). Neu entwickelt wurde von Herrn Gallenkamp eine Leiterplatte, die der Realisierung der folgenden Funktionen dient:

Gemeinsamer Speicher zum Datenaustausch zwischen den Busmastern (Rechnelementen und PC-Schnittstelle)

PC-Schnittstelle (remote interface) für Download und Bedienungseingriff während der Simulation

Bus-Arbiter zur Zugriffskoordination auf den gemeinsamen External Bus.

Genormte Sercos-interface Schnittstelle zum Anschluß einer Steuer- bzw. Regeleinrichtung

Transientenrecorder (4MB DRAM)

Software

Der Simulator dient zur Nachbildung kontinuierlicher (linearer und nichtlinearer) Systeme und ggf. sind auch Zusätze für zeitdiskrete Systeme realisierbar. Nicht realisierbar sind dagegen diskontinuierliche Systeme, die abschnittsweise unterschiedliche mathematische Beschreibungen benötigen, wie z.B. Stromrichterschaltungen, bei denen sich das zugehörige Gleichungssystem im Moment des Schaltens von Ventilen ändert.

Die Beschreibung des nachzubildenden Systems erfolgt anhand eines Blockschaltbildes. Beim derzeitigen Ausbauzustand stehen dem Anwender folgende unterschiedliche Funktionsblöcke zur Verfügung: 14 algebraische Blöcke (SUM, MUL,...) , 8 „histioy“-Blöcke (Integratoren, VZ1, VZ2...) sowie diverse Quellen und Senken. Die Codesequenzen dieser Funktionsblöcke, die neben dem

TMS320C30-Assemblercode noch Angaben zur Zahl der Ein- und Ausgänge sowie die Ausführungsdauer enthält, sind in einer Code-Bibliothek abgelegt.
Die Bedienoberfläche des Simulators läuft auf einem PC, der über eine serielle Schnittstelle mit dem Parallelrechner („PC-Schnittstelle“) verbunden ist. Am PC wird das Blockschaltbild mittels eines handelsüblichen Stromlaufplan -Editors (ORCAD) eingegeben, wobei einer Bild - Bibliothek die speziellen Block - Symbole entnommen werden. Der Stromlaufplan - Editor erzeugt eine Netzliste, die die Struktur des Blockschaltbildes beschreibt. Zusätzlich ist vom Anwender eine Parameterdateizu erstellen.
Im Gegensatz zu den meisten bekannten Simulatoren (- bzw. den damit vergleichbaren freiprogrammierbaren Reglern verschiedener Firmen-) wird zur Laufzeit **keine Interpretation** des Blockschaltbildes ausgeführt, da dieses nicht rechenzeitoptimal ist.
Kernstück der von Herrn Gallenkamp entwickelten Software ist der **Compiler** der aus der Netz- und Parameterliste (Quelle) unter Verwendung der Codebibliothek den optimierten Assenblercode des nachzubildenden Systems erzeugt.

Zunächst wird die Arbeitsweise des Compilers für ein einzelnes Rechenelement beschrieben. Die Erweiterungen für den gesamten Parallelrechner werden später behandelt. Der Compiler erzeugt aus der - das Blockschaltbild repräsentierenden - Netzliste optimalen Assemblercode, der anschließend mittels Assembler, Linker und Locater in ausführbaren Maschinencode umgesetzt wird. Als erster Schritt müssen die Blöcke so sortiert werden, daß bei der Berechnung eines Blockes seine sämtlichen Eingangsgrößen schon berechnet sind. Hierzu sind die Blöcke in drei Klassen eingeteilt: 1. Signalquellen, 2. Signalsenken und 3. Algebraische Blöcke. Ein Block, dessen Ausgangswert während der aktuellen Abtastperiode unabhängig von seinen Eingangsgrößen ist, wird History-Block genannt. Hierzu gehören beispielsweise alle Integratoren. Ein History-Block wird im folgenden als zwei einzelne Blöcke angesehen, nämlich einem History-Front-End, das die Eigenschaften einer Signalsenke hat und einem History-Back-End, welches die Eigenschaft einer Signalquelle hat. Die Netzliste enthält in Textform die Beschreibung des Blockschaltbildes als gerichteten Graphen. Die Knoten dieses Graphen sind die Blöcke, die Kanten Verbindungen zwischen diesen Blöcken. Die gesuchte Abarbeitungsreihenfolge wird durch Traversierung dieses Graphen gefunden. Dazu wird beginnend bei den Senken (Wurzeln der Unterbäume) entgegen der Signalflußrichtung des Blockschaltbildes ein Pfad über die davorliegenden Blöcke bis hin zu einer Quelle aufgebaut. Der dazugehörige Traversierungsalgorithmus ist in sehr kompakter und logisch klar aufgebauter rekursiver Form in Tabelle 4.4 dargestellt. Dabei werden die aus der Code-Bibliothek entnommenen Assembler Codesequenzen in der Reihenfolge beginnend bei der Quelle für alle Knoten (Blöcke) entlang dieses Pfades bis hin zur Senke aneinandergereiht. Der Datenfluß zwischen diesen Blöcken wird durch einen speziellen Stack-Mechanismus realisiert. Dazu wird innerhalb des lokalen RAM des TMS 320C30 ein Speicherbereich reserviert, auf den mittels eines als Stack-Pointer benutzten Adressregisters zugegriffen wird. Um den Datenfluß zu optimieren werden häufig benutzte Sack-Elemente, -das sind die jeweils obersten Elemente des Stacks (der Stack-Top)- nicht in diesem RAM sondern in Prozessorregistern gehalten, so daß das Einlesen vom internen RAM in die Prozessorregister und umgekehrt entfällt. Erst wenn nicht mehr genügend Prozessorregister zur Verfügung stehen werden die einzelnen Register in das interne RAM übernommen. Der eigentliche Stack ist somit in zwei Teile unterteilt: Einen verhältnismäßig kleinen Register-Stack und einen größeren Haupt-Stack, der im internen RAM gehalten wird. Die Zuordnung der Register zu den Stack-Elementen geschieht bereits während der Compilierungszeit und ist während der Laufzeit fest. Es zeigt sich, daß die meisten in der Praxis benutzten Funktionsblöcke in aller Regel nur den Register-Stack benötigen, was einen deutlichen Geschwindigkeitszuwachs bedeutet.

In Kapitel 5 werden dann die zusätzlichen Maßnahmen beschrieben, die zur automatischen Parallelisierung erforderlich sind. Der erste Schritt zur Parallelisierung des Gesamtproblems ist dessen Aufteilung in mehrere kleine Teilprobleme, die sich voneinander unabhängig rechnen lassen und im folgenden atomare Subtasks genannt werden. Ein atomarer Subtask wird im Blockschaltbild einen History-Block z.B. einen Integrator zugeordnet. Der atomare Subtask erstreckt sich von dem jeweiligen History-Block ausgehend, entgegen der Signalflußrichtung über ein algebraisches Verknüpfungsnetz bis zu den Ausgängen der anderen History-Blöcke und/oder Output-Blöcke. Wenn Teile des jeweils vorgeschalteten algebraischen Verknüpfungsnetzes gemeinsam als Eingangsgrößen verschiedener History-Blöcke genutzt werden, dann sind diese (mehreren) History-Blöcke Bestandteil des atomaren Subtasks. Ziel der optimalen Parallelisierung ist es, Modellteile mit enger Verkopplung möglichst auf einem Rechenelement zu rechnen (Minimierung der Kommunikation zwischen den Rechnern) und die einzelnen Rechenelemente möglichst gleichmäßig auszulasten. Zunächst werden Paare aus (je zwei) atomaren Subtasks gebildet und mit einer Ziffer, die die jeweilige Anzahl gegenseitiger Verbindung im Blockschaltbild angibt (connectivity), charakterisiert. Diese Paare werden in einer Liste nach absteigender Folge dieser Connectivity sortiert, d.h. die stark verkoppelten

Paare aus atomaren Subtasks stehen am Anfang dieser Liste und sollten möglichst nicht auf unterschiedliche Rechenelemente verteilt werden. Nach einer anfänglich noch willkürlichen Verteilung der atomaren Subtasks auf die einzelnen Rechenelemente wird dann versucht die atomaren Subtasks mit großen gegenseitigen Abhängigkeiten in wenigen aber größeren Subtasks zu vereinigen. Mit zunehmender Größe und sinkender Anzahl der Subtaskmengen wird es zunehmend schwieriger die entstandenen Substaskmengen unter Berücksichtigung der verfügbaren Rechenzeit auf die Rechenelemente zu verteilen. Ist auf keinem der Rechenelemente genügend Rechenzeitreserve vorhanden um anstelle zweier einzelner Subtaskmengen eine größere einzelne Subtaskmenge zu plazieren, dann wird versucht durch Umsortierung der vorhandenen Subtaskmengen die jeweiligen Rechenzeitreserven der Rechenelemente zu konzentrieren und so auf einem einzelnen Rechenelement eine größere Zeitreserve zu enthalten. Schlägt auch dies fehl, dann muß auf die Zusammenführung der Subtasks verzichtet werden, d.h. diese müssen dann eben trotz erhöhtem Kommunikationsaufwand auf unterschiedlichen Rechenelementen bearbeitet werden. Die dazu notwendigen Prozeduren sind in übersichtlicher und logisch sauber aufeinander aufbauender Form in den Tabellen 5.1 bis 5.4 zusammengestellt.

Um der gesamten Einrichtung Echtzeitverhalten zu geben, muß natürlich die Rechenzeit des am stärksten belasteten Rechenelementes kürzer als die Simulations-Abtastperiode sein. Diese Simulations-Abtastperiode wird in vier Phasen unterteilt. Die erste und wichtigste Phase ist die Rechenphase. Da im allgemeinen nicht alle Rechenelemente gleichmäßig auslastbar sind, schließt sich daran eine erste Synchronisationsphase an. Hier wird so lange gewartet bis alle Rechenelemente mit der Rechenphase fertig sind. Dann kommt die Datenaustauschphase zwischen den Rechenelementen. Auch diese kann unterschiedlich lang sein, so daß sich an ihr Ende wiederum eine Synchronisationsphase anschließt. Alle vier Phasen zusammen müssen in die Simulations-Abtastperiode passen. Falls dies nicht gegeben ist müssen zur Lösung der Aufgabe mehr Rechenelemente parallel eingesetzt werden.
Zum Abschluß des Software-Teiles der Arbeit wird der erzielbare Rechenleistungsgewinn bei zwei bzw. drei Rechenelementen gegenüber der Rechenleistung eines Rechenelementes verglichen, mit dem Ergebnis, daß bei einer Abtastperiode von 100 µs mit drei Rechenelementen ungefähr die 2,5-fache Rechenleistung des Einzelrechnersystems nutzbar ist, der Rest ist „Overhead“.

Im letzten Kapitel wird ein kurzes Anwendungsbeispiel, das die Regelung de Walzspaltes an einem Kaltwalzgerüst beschreibt, vorgestellt. Es handelt sich um eine nichtlineare Regelstrecke 7. Ordnung. Das Glied mit der höchsten Eigenfrequenz in diesem Prozeß ist das Servo-Ventil, dessen Resonanzfrequenz bei 240 Hz liegt. Mit einer Abtastperiodendauer von t=100 µs werden ausreichend stabile Integrationsergebnisse erzielt. Es zeigt sich, daß dieses Beispiel ohne jede Schwierigkeit bereits auf einem der drei vorhandenen Rechenelemente abgearbeitet werden kann wobei dieses eine Rechenelement nur zu 35% ausgelastet ist. Um die volle Leistungsfähigkeit des Echtzeitsimulators auszuloten müssen also wesentlich komplexere Regelstrecken herangezogen werden.

P. Mutschler