Christian-Albrechts-Universität zu Kiel

Diplomarbeit

SyncCharts in C auf Multicore

Niclas Stephan Köser

20. Oktober 2010

Institut für Informatik
Lehrstuhl für Echtzeitsysteme und Eingebettete Systeme

Prof. Dr. Reinhard von Hanxleden

betreut durch:
Prof. Dr. Reinhard von Hanxleden

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

_____

Abstract

Commonly, reactive systems are implemented as single-core systems. Distributed real-time systems communicate over asynchronous links and are therefore not synchronous. In the field of embedded systems, symmetric multicore CPUs are still comparatively uncommon and rarely programmed with synchronous languages. This thesis compares multicore platforms for embedded systems and exposes those with very deterministic behavior. The programming language Synchronous C (SC), for which code can be generated from SyncCharts, is being extended by mechanisms for communication and synchronization. This permits a reactive, error-free execution with little overhead on a multicore CPU. Implementations for the Precision Timed Architecture (PRET) and a Linux environment are presented. Moreover, principles of translating SyncCharts to SC on multicore, which exploit the inherent concurrency of a SyncChart, are explained.

Reaktive Systeme sind üblicherweise Singlecore-Systeme. Verteilte Echtzeitsysteme verlieren ihre synchronen Eigenschaften, da sie asynchron kommunizieren. Symmetrische Multicore-Systeme sind im Bereich eingebetteter Systeme noch nicht weit verbreitet und werden selten mit synchronen Sprachen programmiert. In dieser Arbeit werden verschiedene Multicore-Plattformen für eingebettete Echtzeitsysteme verglichen und Multicore-Prozessoren mit besonders hohem Grad an Determinismus herausgestellt. Die Programmiersprache Synchronous C (SC), die direkt aus SyncCharts übersetzt werden kann, wird um Synchronisations- und Kommunikationsmechanismen erweitert, die eine reaktive, fehlerfreie Multicore-Ausführung mit geringem Overhead erlauben. Implementierungen für den Precision Timed Architecture (PRET) und eine Linux-Umgebung werden vorgestellt und Ansätze zur Übersetzung von SyncCharts, die die inhärente Nebenläufigkeit eines SyncCharts ausnutzt, erläutert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listingverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ALU	arithmetisch-logische Einheit (engl. *Arithmetic Logic Unit*)
ARPRET	Auckland Reactive PRET processor
BCC	LEON Bare-C Cross-Compiler System
BSD	Berkeley Software Distribution
CAN	Controller Area Network
CFSM	Co-design Finite State Machine
CPU	Prozessor (engl. *Central Processing Unit*)
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DSP	digitaler Signalprozessor
EIB	Element Interconnect Bus
ELF	Executable and Linking Format
EMPEROR	Embedded MultiProcessor supporting Esterel Reactive OpeRations
FIFO	First In – First Out
FPU	Fließkommaeinheit (engl. *Floating Point Unit*)
FPGA	Field Programmable Gate Array
FSL	Fast Simplex Link
GALS	global asynchrones, lokal synchrones System
GDB	GNU Debugger
GCC	GNU Compiler Collection
GNU	GNU's Not Unix
GPL	General Public License

GPU	Graphics Processing Unit
GPGPU	General-Purpose computation on Graphics Processing Units
GUI	Benutzeroberfläche (engl. *Graphical User Interface*)
I²C	Inter-Integrated Circuit
IDE	integrierte Entwicklungsumgebung (engl. *Integrated Development Environment*)
JTAG	Joint Test Action Group (IEEE-Standard 1149.1)
KEP	Kiel Esterel Processor
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
LCR	Le Lann, Chang, Roberts (ein Algorithmus)
LDT	Long-Duration-Task
LLVM	Low Level Virtual Machine
MFLOPS	Million Floating Point Operations per Second
MIPS	Million Instructions per Second
MIPS	Microprocessor without Interlocked Pipeline Stages
MPPA	Massively Parallel Processor Array
MPSoC	Multiprocessor System on a Chip
MUTEX	Mutual Exclusion
NIST	National Institute of Standards and Technology
NOP	No Operation (auch *No Operation Performed*)
NUMA	Non-Uniform Memory Architecture
OC	Object Code
PIC	Programmable Intelligent Computer
PID	Process ID (eine Thread-ID in SC)
PCI	Peripheral Component Interconnect
POSIX	Portable Operating System Interface
PPE	PowerPC Processing Element

PRET	Precision Timed Architecture
PRET-C	Precision Timed C
PROM	Programmable Read-Only Memory
RePIC	Reactive PIC
RISC	Reduced Instruction Set Computer
RTAI	Real-Time Application Interface
RTOS	Real-Time Operating System
SC	Synchronous C
SCmc	Synchronous C auf Multicore
SHIM	Software/Hardware Integration Medium
SIMD	Single Instruction, Multiple Data
SMP	Symmetric Multiprocessing
SMT	Simultaneous Multithreading
SoC	System on a Chip
SPARC	Scalable Processor Architecture
SPE	Synergistic Processing Element
SPI	Serial Peripheral Interface Bus
SRAM	Static Random Access Memory
TCU	Thread-Control-Unit
USB	Universal Serial Bus
VHDL	Very High Speed Integrated Circuit Hardware Description Language
WCET	Worst Case Execution Time
WCET	Worst Case Reaction Time

1. Einleitung

Der Begriff des reaktiven Systems wurde 1984 von Harel und Pnueli [33] geprägt. Während ein transformatives System nach der Berechnung eines Ergebnisses zu gegebenen Eingaben terminiert und als „Black Box“ wahrgenommen wird, interagiert ein reaktives System permanent mit seiner Umgebung, oft durch eine Vielzahl von Ein- und Ausgaben. Es erscheint daher eher wie ein „schwarzer Kaktus“.

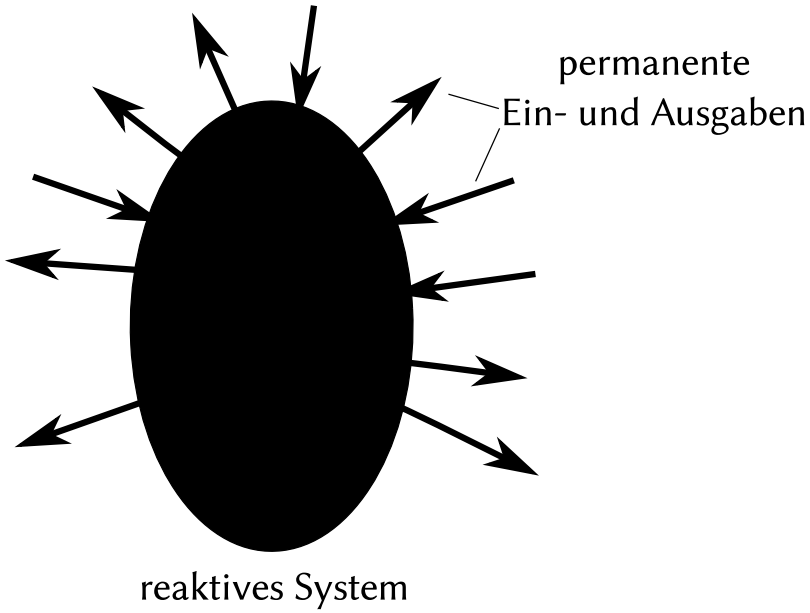


Abbildung 1.1.: Harels schwarzer Kaktus

Im Gegensatz zu reinen interaktiven Systemen müssen reaktive Systeme aber Echtzeitanforderungen erfüllen [61, Lecture 01]. Das heißt, ihre Umgebung erwartet nicht nur ständige Interaktion, sondern gibt dabei auch die Reaktionsgeschwindigkeit vor [5]. Sie sind oft im Bereich eingebetteter Systeme anzutreffen, also solcher Systeme, die nicht unbedingt als Computer wahrgenommen werden, weil sie innerhalb von anderen Systemen operieren und Steuerungsaufgaben übernehmen. Die meisten industriellen Echtzeitsysteme wie Kontroll-, Überwachungs- und signalverarbeitende Systeme zählen zu den reaktiven Systemen [15].

Reaktive Systeme müssen darüber hinaus deterministisch sein, das heißt, zu gleichen Eingaben zuverlässig gleiche Ausgaben erzeugen [63, Lecture 02 & 03]. Dadurch werden sie verifizierbar und vorhersehbar [61, Lecture 01]. Die Steuerung eines Kraft-

werks, eines Flugzeugs oder eines medizintechnischen Gerätes muss nicht nur schnell, sondern auch beweisbar korrekt sein.

Der Forderung nach deterministischem Verhalten laufen mehrere Dinge zuwider:

- Nebenläufigkeit und Preemption sind notwendige Konzepte, ohne die die Modellierung eines reaktiven Systems nicht komfortabel möglich ist. Sie bringen aber auch Komplexität und Unvorhersehbarkeiten mit sich.

- Eingebettete reaktive Systeme sind oft verteilte Systeme [27], weil die Komponenten näher an Sensoren und Aktuatoren sein müssen oder Fehlertoleranz oder Leistungsfähigkeit erhöht werden sollen.

- Moderne Prozessoren wurden außerdem zugunsten der mittleren Ausführungszeit regelrecht hochgezüchtet, und zwar mit komplexen Optimierungen und Erweiterungen, die ihr Verhalten schwer vorhersehbar machen [38].

Zur Programmierung reaktiver Systeme eignen sich synchrone Programmiersprachen wie Esterel [12] oder SyncCharts [5], die Nebenläufigkeit und Preemption *deterministisch* beherrschen. Sie abstrahieren unter Annahme der synchronen Hypothese [51] vom zeitlichen Verhalten des Zielsystems, so dass zunächst unabhängig von dessen Hardware programmiert und analysiert werden kann [63, Lecture 03].

Diese Arbeit beschäftigt sich mit Synchronous C (SC), einer makro-basierten Erweiterung der Programmiersprache C. Sie erlaubt durch prioritätenbasiertes Scheduling deterministische Nebenläufigkeit mit leichtgewichtigen Threads [62]. SC kann auch als Zielsprache für die Übersetzung von SyncCharts dienen [3].

Das synchrone Modell berücksichtigt verteilte Systeme nicht [27]. Da kausale Abhängigkeiten berücksichtigt werden müssen, kann aus Design-Parallelität nicht unbedingt auf Laufzeit-Parallelität geschlossen werden. Die Komponenten eines verteilten Systems haben in der Regel getrennte Speicher, agieren autonom [14, S. 523ff] und kommunizieren asynchron über möglicherweise unzuverlässige Verbindungen. Dies führt zu unbekannten Verzögerungen [60, Lecture 02] und erfordert spezielle Kommunikationsprotokolle. Daher existieren überwiegend Ansätze zur Umsetzung von synchronen Programmen in global asynchrone, lokal synchrone Systeme (GALS) und zur Ratendesynchronisierung.

Multicore-Prozessoren leiden hingegen durch die enge Kopplung der Kerne nur eingeschränkt unter den Problemen verteilter Systeme. Die Kommunikation ist im Vergleich extrem schnell und vor allem zuverlässig, so dass eine Synchronisation sinnvoll erscheint.

Andererseits verschlimmern Multicore-Prozessoren das Problem des indeterministischen Verhaltens, weil mehrere Kerne um gemeinsam genutzte Ressourcen konkurrieren oder sich direkt gegenseitig beeinflussen können. Neuartige Ansätze wie die Precision Timed Architecture (PRET) [44] sollen dem Problem entgegenwirken, indem sie das zeitliche Verhalten eines Prozessors vorhersehbar machen.

1.1. Aufgabenstellung

Ziel dieser Arbeit ist die Erkundung von Möglichkeiten zur parallelen Ausführung von SC-Programmen. Der Schwerpunkt liegt dabei auf der möglichen Leistungssteigerung durch Nutzung der inhärenten Parallelität in SyncCharts. SC muss also so um Kommunikations- und Synchronisationsmechanismen erweitert werden, dass keine Inkohärenzen auftreten und das Verhalten deterministisch bleibt.

Da SC darauf ausgelegt ist, ohne ein Betriebssystem ausgeführt zu werden, hängen die Möglichkeiten hierzu von der konkreten Hardware ab. Eine Betrachtung verschiedener Multicore-Plattformen ist daher notwendig.

1.2. Überblick

In Kapitel 2 werden zunächst verwandte Arbeiten, insbesondere Verfahren zur Verteilung über asynchrone Netzwerke, vorgestellt. Kapitel 3 erklärt synchrone Sprachen, speziell SyncCharts, und die synchrone Hypothese, bevor SC eingehend betrachtet und die Synthese von SC-Code aus SyncCharts vereinfacht erklärt wird. Multicore-Plattformen werden in Kapitel 4 ausführlich behandelt und ein geeigneter Prozessor ausgewählt. Kapitel 5 und 6 sind dem signalbasierten und dem prioritätenbasierten Ansatz zur Synchronisation sowie den dabei entstehenden Problemen und den Implementierungen gewidmet. Experimentelle Ergebnisse und mögliche zukünftige Arbeiten werden in Kapitel 7 und 8 vorgestellt.

2. Verwandte Arbeiten

Mit dem Compiler von Torsten Amende [3] existiert ein Plugin für das KIELER-Framework[1], das SyncCharts direkt in C übersetzen kann. Die Ausführung in SC [62] erfolgt sequentiell, Ansätze zur gleichzeitigen Abarbeitung von Threads existieren bislang nicht.

Bisherige Arbeiten zur Verteilung von synchronen Programmen beschäftigen sich überwiegend mit der Verteilung über asynchrone Netze und der nachträglichen Resynchronisierung, nicht mit der möglichen Leistungssteigerung durch parallele Ausführung. Girault gibt in seinem „Survey“ [27] einen Überblick. Auf die dort genannten Arbeiten wird im Folgenden zuerst eingegangen. Danach werden weitere Arbeiten und Projekte wie SHIM, EMPEROR, der reaktive Prozessor KEP und die Sprache PRET-C betrachtet.

ocrep und Saxo-RT

Für das ocrep-Tool [15] wird Zwischencode im OC-Format (Object Code) benötigt, welcher aus Lustre und Esterel erzeugt werden kann. OC beschreibt einen endlichen Automaten, der durch ocrep auf allen Ausführungsorten (*Locations*) repliziert wird. Anhand einer Vorgabe, auf welcher Location welche Ein- und Ausgaben verarbeitet werden sollen, wird festgestellt, welche Codezweige nicht benötigt werden, und diese werden deaktiviert. Um die Signalabhängigkeiten aufzulösen, werden Sende- und Empfangskommandos eingefügt. Bei Bedarf sorgen „Dummy“-Sendungen dafür, dass das Programm synchron bleibt.

Girault, Nicollin und Pouzet haben das Verfahren erweitert [29], um eine Desynchronisierung von Long-Duration-Tasks (LDT) zu erreichen, also solchen Aufgaben, die deutlich länger benötigen als andere, und dadurch die Taktrate des Systems begrenzen. Hierzu muss jedoch im Vorfeld bekannt sein, mit welcher Frequenz diese maximal ausgeführt werden.

Dem ocrep-Verfahren sind durch die kombinatorische Explosion bei der Erzeugung von Code im OC-Format Grenzen gesetzt. Ein von der France Telecom R&D entwickelter Esterel-Compiler namens Saxo-RT [17], der den ocrep-Algorithmus ebenfalls anwenden kann, verwendet ein anderes Zwischenformat ohne kombinatorische Explosion.

[1] `http://www.informatik.uni-kiel.de/rtsys/kieler`

screp

Screp [28] verarbeitet das Zwischenformat SC (Sequential Circuit), nicht zu verwechseln mit Synchronous C, welches ein Programm als azyklischen Schaltkreis beschreibt[2]. Dieser besteht, vereinfacht gesagt, aus Teilnetzen, die Eingaben zur Verfügung stellen, Ausgaben anzeigen, Bedingungen prüfen oder in einer Tabelle definierte *Actions* wie Variablenzuweisungen oder externe Prozeduraufrufe auslösen. Letztere werden als *Action-Netze* bezeichnet. Screp weist jede Action genau einer Location zu. Action- und Ausgabe-Netze, die nicht zur jeweiligen Location gehören, werden in normale Netze ohne Außenwirkung umgewandelt, da ihre Ergebnisse möglicherweise lokal benötigt werden. Eingabe-Netze werden durch Eingabesimulationsblöcke ersetzt, die die Abhängigkeiten von Signalen anderer Locations auflösen, dabei aber unnötige Kommunikation verhindern sollen. Die verschiedenen entstehenden Netze können, nachdem Kommunikationsfunktionalität eingefügt wurde, verteilt werden. Screp partitioniert also anhand der Daten und repliziert den Kontrollteil des Schaltkreises.

Co-design Finite State Machines in POLIS

Girault erwähnt eine Arbeit von von Berry und Sentovich [13], die aus Esterel generierte Schaltkreise in POLIS implementieren. POLIS ist ein Framework für gemeinsames Hard- und Software-Engineering (engl. „Co-design“), in dem mit Co-design Finite State Machines (CFSMs), asynchron kommunizierenden endlichen Automaten, modelliert wird[3]. Bei diesem Verfahren werden Gatter in Clustern zusammengefügt, die Granularität kann freier bestimmt werden. Das CFSM-Netzwerk als ganzes bildet das Verhalten des ursprünglichen synchronen Programms durch Kommunikation über die innere Zustände der Automaten ab. Der Kommunikationsaufwand ist also höher als bei screp, wo jede Komponente Werte publiziert, und die Kohärenz durch die Replikation der Kontrollstrukturen gewährleistet wird.

SynDEx

Das SynDEx-Tool [54] ist für massiv-parallele Multicomputer-Systeme entwickelt worden. Es erwartet als Eingabe ein Programm in Form eines Datenflussalgorithmus', wie er vom Signal-Compiler erzeugt werden kann, und eine Beschreibung der Hardware. Beim Verteilen des Programms versucht SynDEx, den Kommunikationsoverhead zu minimieren. Dabei handelt es sich um ein NP-vollständiges Problem, welchem mit Heuristiken begegnet wird.

Alle diese Systeme versuchen, Entwicklern die Arbeit abzunehmen, ein verteiltes System zu designen. Stattdessen soll ein zentrales, synchrones Programm, das leicht

[2] Ausführlich illustriert ist Screp in Vortragsfolien von Girault und Ménier: `ftp://ftp.inrialpes.fr/pub/bip/pub/girault/Presentations/emsoft02.pdf.gz`

[3] `http://embedded.eecs.berkeley.edu/research/hsc/`

verifiziert werden kann, kompiliert und nachträglich auf vorgegebene Locations verteilt werden.

Ocrep, Saxo-RT, screp und die POLIS-Methode übersetzen allerdings nur Esterel und versuchen darüber hinaus, Abläufe über asynchrone Kommunikationskanäle zu synchronisieren oder ein global asynchrones, lokal synchrones System (GALS) zu erzeugen. GALS sind Systeme, in denen endliche Automaten sich wie synchrone Systeme verhalten, jedoch ansynchron kommunizieren[4]. Muttersbach et al. beschreiben sogar die Umsetzung des GALS-Konzeptes in Hardware [49]. Ziel dieser Arbeit ist hingegen, die Ausführung von Synchronous C auf einem Multicore-System zu ermöglichen. Hier kann von einer ausreichend engen Kopplung ausgegangen werden, die ein global synchrones System gestattet. Girault erwähnt außerdem einen Ansatz, mittels eines die Datenabhängigkeiten berücksichtigenden Schedulers Lustre-Programme über eine Time-Triggered-Architecture zu verteilen [27, Abschnitt 4.5]. Die Signalabhängigkeiten werden bei der Übersetzung von SyncCharts nach SC [3] bereits in Thread-Priorisierung umgesetzt.

Aubry und Le Guernic [6] halten die Desynchronisierung synchroner Programme für paradox. Sie schlagen vor, stattdessen die Semantik zu minimieren, und so die Einschränkungen auf Kosten der Synchronizität zu lockern, um die verteilte Ausführung von vornherein zu erleichtern. Allerdings konzentriert sich ihre Arbeit auf Datenflusssprachen. Eine Lockerung der Synchronizität würde außerdem die Semantik von SyncCharts brechen und brächte für ein Multicore-System aus oben genannten Gründen keinen Vorteil. Allenfalls als nachträgliche Optimierung der Ausführung von Synchronous C auf Multicore (SCmc) wäre eine asynchrone Ausführung denkbar, diese müsste aber die Korrektheit der Spezifikation erhalten.

Die Arbeiten zur Endo- und Isochronizität von Benveniste et al. [9, 52] beschäftigen sich vereinfacht gesagt mit der Frage, wie ein synchrones Programm ohne Verlust seiner Semantik zerlegt werden kann. Die Kommunikation zwischen den Teilen soll so weit wie möglich reduziert werden, um die nebenläufige Ausführung nicht zu behindern, während eine Resynchronisierung unabhängiger Teile die Korrektheit erhält[5]. Hier ist der Ausgangspunkt allerdings die deklarative Sprache Signal. Die Konzepte sind für ein Multicore-System außerdem zu komplex und haben ebenfalls die Konstruktion eines GALS zum Ziel. In dieser Arbeit soll hingegen der synchrone Bereich über mehrere Prozessorkerne ausgedehnt werden.

Das SynDEx-Tool erscheint vielversprechend, schließlich kann ein Multicore-System als Sonderfall eines Multicomputers angesehen werden. Dies gilt insbesondere dann, wenn getrennte Speicher zur Verfügung stehen. Statisches Scheduling und das Erfordernis, die Eingabe in einer Datenflusssprache zu liefern, stehen einer Anwendung bei SyncCharts oder SC jedoch im Wege.

Herlihy empfiehlt, Wissen aus dem Bereich verteilter Systeme im Multicore-Bereich zu nutzen und weiterzuentwickeln [35], setzt dabei jedoch auf Transactional Memo-

[4] http://www.ida.liu.se/~TDTS07/lectures/lect7.frm.pdf

[5] Siehe hierzu auch die Vortragsfolien von Dumitru Potop-Butucaru: http://www-verimag.imag.fr/PEOPLE/Nicolas.Halbwachs/SYNCHRON03/Slides/potop.ppt

ry, ein Konzept, das bisher noch nicht in Hardware umgesetzt wurde, zu weiteren Indeterminismen führt und daher im Kontext synchroner Systeme keinen Nutzen verspricht.

Software/Hardware Integration Medium (SHIM)

Die in dieser Arbeit verfolgten Ansätze ähneln in gewisser Weise denen von Vasudevan und Edwards [59], die bereits Multicore-Umgebungen im Blick hatten, als sie eine vom Compiler eingefügte Synchronisation zwischen nebenläufigen Threads vorschlugen. Ihre Lösung erlaubt es dem Programmierer, Fehler zu machen, garantiert aber ein wiederholbares Verhalten.

Edwards et al. argumentieren [41], dass es, selbst wenn die CPU ein bestimmtes zeitliches Verhalten garantiert, schon wegen der Eingaben unmöglich sei, präzise Vorhersagen über das Timing zu machen. Wiederholbares Timing sei wichtiger und leichter erreichbar, zudem mache es Testergebnisse erst brauchbar. Die Sprache, die aus dem SHIM-Projekt [21] hervorgegangen ist, beschreibt daher asynchrone nebenläufige Threads, deren Ein- und Ausgaben nicht durch das Scheduling beeinflusst werden. Der Compiler erzeugt sequentiellen Code und fügt Synchronisations-Anweisungen in den Code ein [59], die Kommunikation findet über Kanäle mit blockierendem Senden statt.

Im Rahmen des SHIM-Projekts entstand letztendlich auch ein Backend für einen Multicore-Prozessor, den Cell [50]. Bei diesem koordiniert der PowerPC-Kern[6] die Kommunikation der anderen Kerne, der sogenannten Synergistic Processing Elements (SPE) [58].

Die zentrale Koordinierung durch einen Kern wird in den prioritätsbasierten Ansätzen in dieser Arbeit ebenfalls angewandt. SHIM berücksichtigt jedoch keine kausalen Abhängigkeiten, man programmiert also nicht synchron [62].

Precision Timed C (PRET-C)

PRET-C, eine synchrone Erweiterung von C, ist auf bessere zeitliche Vorhersagbarkeit auf „General Purpose Processors“, also Mehrzweck-CPUs, die nicht auf eine Anwendung spezialisiert sind, ausgerichtet [4]. Auf einem angepassten Prozessor sind sogar präzise Vorhersagen möglich [53].

PRET-C besitzt ein par-Statement, das logische Nebenläufigkeit ausdrückt. Wie bei SC sind die Threads sehr leichtgewichtig, zur Laufzeit werden sie aber sequentiell und in der Reihenfolge ihres Vorkommens im Quellcode ausgeführt. Der Programmierer muss also kausale Abhängigkeiten selbst auflösen. Ansätze für Nebenläufigkeit zur Laufzeit bieten die Veröffentlichungen zu PRET-C nicht.

[6] PowerPC-Architektur siehe `http://de.wikipedia.org/w/index.php?title=PowerPC&oldid=79678169`

EMPEROR

Ähnlichkeit besteht auch zwischen dem EMPEROR-Projekt [20] und der in dieser Arbeit beschriebenen signalbasierten Synchronisation. EMPEROR steht für „Embedded MultiProcessor supporting Esterel Reactive OpeRations“. Grundlage des EMPEROR-Systems sind modifizierte PIC-CPUs[7], die durch ihren um einige Esterel-Kommandos erweiterten Befehlssatz eine höhere Ausführungsgeschwindigkeit erreichen.

Der EMPEROR-Compiler nutzt mittels eines Kontrollflussgraphen die im Esterel-Code ausgedrückte Parallelität aus und zerlegt ihn in Threads, anstatt ihn zu sequentialisieren. Die Threads werden unter Replikation eines Teils der Kontrollflusslogik auf verschiedene „Computing Locations“, das heißt CPUs, verteilt [66].

Die CPUs sind über in Hardware realisierte Kommunikations- und Synchronisierungsmechanismen gekoppelt. Neben einer Barrier zur Synchronisierung der Ticks existiert für Signale ein eigener globaler Registersatz. Die Signale sind, wie bei der in dieser Arbeit verwendeten Certainty-Matrix, dreiwertig kodiert, der Zustand „unbekannt“ ($\bot$) existiert und ist prüfbar. Er ist dann gegeben, wenn ein Signal bislang nicht emittiert wurde, in einer Liste von potentiellen Emittern aber noch solche stehen, die es im laufenden Tick noch emittieren könnten. Jede CPU führt einen Wrapper aus, der die ihr zugewiesenen Threads umlaufend schedulet, bis diese ihren Tick abgeschlossen haben. Kann ein Thread wegen eines Signals mit Zustand $\bot$ nicht weiterlaufen, wird der nächste ausgewählt. Haben alle Threads ihren Durchlauf abgeschlossen, so erreicht die CPU die Tick-Barrier.

EMPEROR ist das einzige dem Autor bekannte Projekt, das auf diese Art versucht, mehrere Prozessorkerne mit einer synchronen Programmiersprache nutzbar zu machen. Allerdings ist im Embedded-Bereich neben der Geschwindigkeit des Systems oft auch dessen Energieeffizienz relevant. Es bleibt unklar, mit welchem Energieaufwand der Geschwindigkeitsvorteil erkauft wird. Zwar ist bemerkenswert, dass selbst bei Berrys „Runner“-Beispiel [10, S. 37] eine höhere Geschwindigkeit gegenüber der Ausführung auf einem einzelnen Kern erreicht wird. Doch zum einen müssen hier größere Teile der Kontrollflusslogik repliziert werden. Zum anderen kommt die permanente Ausführung von Code, solange keine eindeutige Aussage über ein Signal getroffen werden kann, Busy-Waiting gleich. Busy-Waiting ist sehr energieaufwändig, wenn zugunsten schneller Reaktionszeiten keine „Schlaf“- oder Energiesparfunktionen genutzt werden. Die rotierende Ausführung von Threads erschwert darüber hinaus Vorhersagen über das innere Verhalten des Systems, und es wird ausschließlich Esterel-Code ausgeführt. Die zur Compile-Zeit bekannten Abhängigkeiten werden nicht genutzt.

KEP

Der Kiel Esterel Processor (KEP) ist ein multithreading-fähiger reaktiver Prozessor, dessen prioritätenbasierten Scheduling-Mechanismus SC übernommen hat [62]. Für

[7] http://en.wikipedia.org/w/index.php?title=PIC_microcontroller&oldid=386813140

den KEP kann Assemblercode aus Esterel [42] und SyncCharts [55] erzeugt werden. Der Compiler *smakc!* erhält dabei die Struktur des SyncCharts. Ansätze zur gleichzeitigen Ausführung mehrerer Threads existieren bislang nicht.

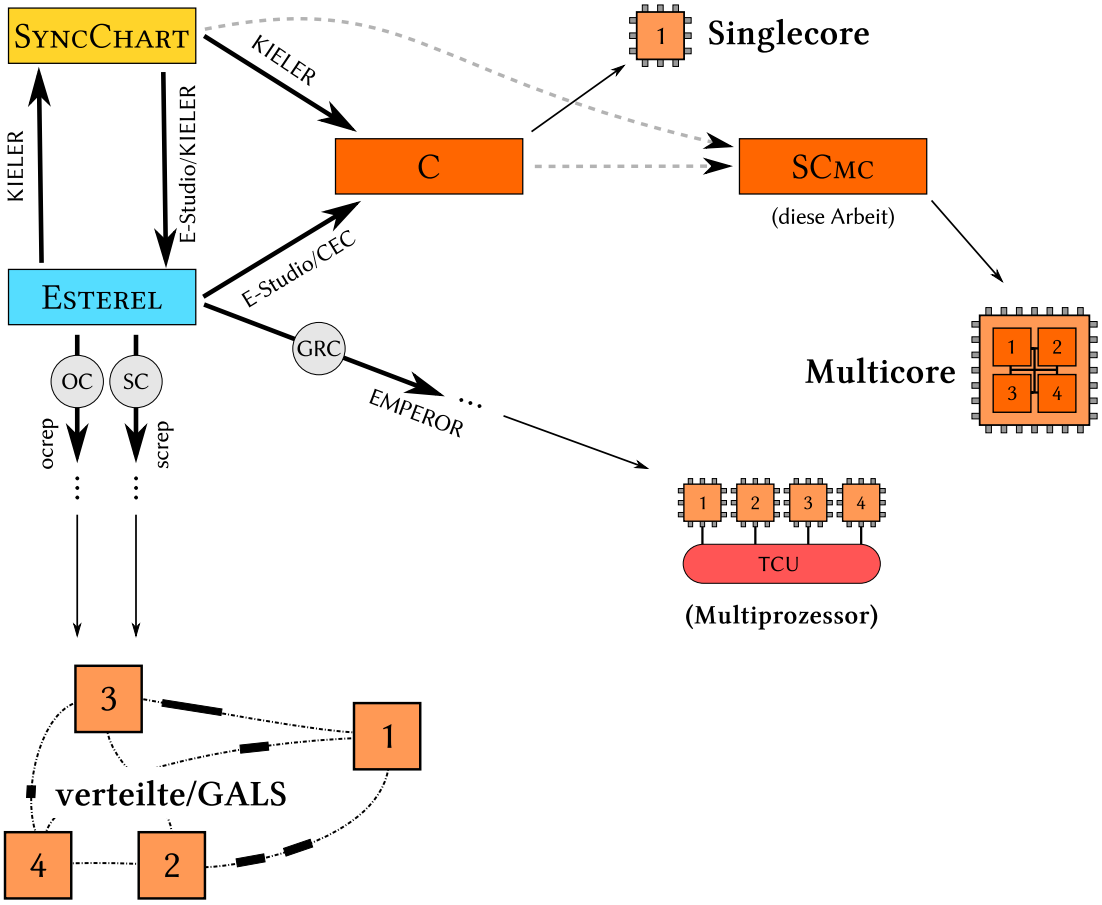


Abbildung 2.1.: Sprachen, Synthesewege und Domänen

Abbildung 2.1 zeigt eine schematische Übersicht der wichtigsten Projekte in ihren Domänen Singlecore, Multicore und Multiprozessor sowie GALS. Programmiersprachen sind eckig, Zwischencodeformate rund dargestellt. Entlang der dicken Pfeile sind Synthesen möglich.

Die Barriers, die die Implementierungen in dieser Arbeit nutzen, entsprechen einfachen Sonderfällen des Beta-Synchronizers [7].

3. Synchrone Sprachen, SyncCharts und SC

In diesem Kapitel werden synchrone Sprachen und die synchrone Hypothese erläutert. Der graphischen Modellierungssprache *SyncCharts* ist ein eigener Abschnitt gewidmet. Im Anschluss daran werden die Grundlagen der Sprache Synchronous C (SC), die C um Signale, Nebenläufigkeit, Hierarchie und Preemption erweitert, erläutert. Zuletzt wird der Compiler, der SC-Code aus SyncCharts synthetisiert, vereinfacht erklärt.

3.1. Synchrone Sprachen

Synchrone Sprachen sind Programmiersprachen, die auf der synchronen Hypothese basieren, die die drei folgenden Annahmen umfasst:

Zero Delay: Reaktionen benötigen keine Zeit und sind atomar [61, Lecture 02].

Perfekte Synchronizität: Auch aus Sicht der einzelnen Teile des Systems benötigen die anderen Teile keine Zeit zur Berechnung ihrer Reaktion [12].

Multiform Notion of Time: Die physikalische Zeit wird als normale Eingabe betrachtet [12].

Die Zero-Delay-Annahme bedeutet, dass Reaktionen in Bezug auf die Umgebung keine Zeit benötigen. In der Praxis heißt das, die Reaktionsgeschwindigkeit des Systems muss kleiner sein als die der Umgebung des Systems[1]. Aus der Umgebung ist dann keine Verzögerung wahrnehmbar. Die Ausführung findet in *Ticks* statt, also diskreten Zeitschritten, und alles, was während eines Ticks geschieht, wird als gleichzeitig angenommen. Eingaben ändern sich also auch nicht während der Berechnung der Ausgaben.

Perfekte Synchronizität bedeutet, dass alle berechneten Ausgaben sofort auch als Eingaben zur Verfügung stehen, und zwar für alle Teile des Systems. Alle Teile nehmen sich gegenseitig und die Umgebung gleich wahr. Aus dieser und der Zero-Delay-Annahme ergibt sich, dass im System nur kausale Abhängigkeiten existieren, jedoch keine zeitlichen. Annahmen über die Reihenfolge, in der Teilreaktionen berechnet

[1] Vortragsfolien von Pascal Raymond `http://www-verimag.imag.fr/~raymond/edu/eng/intro-a.pdf`

werden, sind nicht gestattet, ebensowenig zyklische Abhängigkeiten innerhalb eines Ticks.

Wenn eine Reaktion des synchronen Systems Zeit benötigt, dann nur, weil dies explizit so ausgedrückt wurde. Dabei unterscheidet sich die Wahrnehmung der Zeit nicht von anderen Eingaben. Die Zeit kann also verschiedene Formen annehmen. Zum Beispiel kann eine Zeitspanne in Sekunden ausgedrückt werden, aber auch als Strecke [12].

Charakteristisch für synchrone Sprachen ist außerdem das Signalkonzept: Die Kommunikation mit der Umgebung und zwischen nebenläufigen Teilen des Systems findet über *Signale* statt, die ABSENT oder PRESENT sein können. Wenn ein Signal nicht innerhalb eines Ticks *emittiert* oder *gesendet* wird, dann ist es ABSENT, andernfalls während des gesamten Ticks PRESENT. Signale sind, wenn nicht explizit anders deklariert, immer global sichtbar.

Die synchrone Hypothese erlaubt es, vom zeitlichen Verhalten des zu implementierenden Zielsystems zu abstrahieren. Programmiert wird unabhängig von der Hardware, mit der das System realisiert wird [63, Lecture 03]. Dadurch, dass die Logik des Systems von implementierungsabhängigen Details getrennt wird, werden Analyse, Verifikation und Fehlersuche im Programm erleichtert [12].

Wird ein in einer synchronen Programmiersprache geschriebenes Programm kompiliert, so können kausale Abhängigkeiten in eine feste Ausführungsreihenfolge übersetzt werden. Zyklische kausale Abhängigkeiten sind ausgeschlossen. Eine Kausalitätsanalyse stellt sicher, dass der Status jedes Signals, das geprüft wird, vor der Prüfung festgelegt wurde [51]. Dies ermöglicht es, auch auf den Status ABSENT zu prüfen und darauf zu reagieren. Außerdem ist so die reale Ausführungszeit jedes Ticks beschränkt. Somit kann die tatsächliche Worst Case Execution Time (WCET) oder Worst Case Reaction Time (WCET) [53] des Systems in Abhängigkeit von den Eigenschaften der Hardware bestimmt werden. Abbildung 3.1 stellt dies schematisch dar. Die Abläufe im Inneren des reaktiven Systems, dargestellt als Zustandstransitionen, mögen in der Realität eine gewisse Zeit benötigen, aber diese ist begrenzt und bestimmbar.

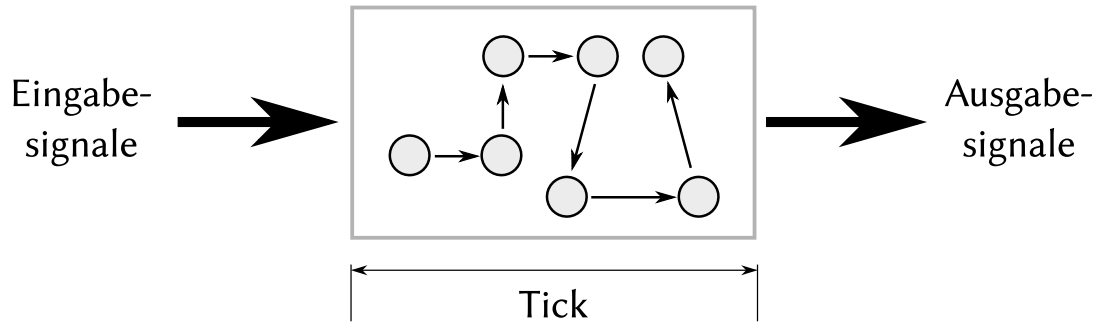


Abbildung 3.1.: Ein Tick

Die meisten reaktiven Systeme können in Teilsysteme zerlegt beschrieben werden [12]. Viele Sprachen unterstützen die modulare Beschreibung, indem sie den Ausdruck von Nebenläufigkeit gestatten und Preemption ermöglichen. Bei nicht synchronen Sprachen bringen beide Konzepte jedoch Komplexität und Unvorhersehbarkeiten mit sich. Synchrone Sprachen wie *Esterel* [11] ermöglichen Nebenläufigkeit und Preemption hingegen deterministisch.

Listing 3.1 zeigt das synchrone Programm ABRO in Esterel. Das Beispiel enthält zwei nebenläufige Threads, die beide nur aus je einer Zeile bestehen: `await A` in Zeile 6 wartet auf ein Signal `A`, `await B` in Zeile 8 auf `B`. Liegt das Signal an, so terminiert `await`. Andernfalls ist der Thread für den Tick mit seiner Ausführung fertig und prüft im nächsten Tick erneut. Sind beide Threads terminiert, weil ihr jeweiliges Signal anlag, so wird in Zeile 10 das Ausgabesignal `O` gesendet. Liegt jedoch das Signal `R` an, so werden die in Zeile 4 beginnende Schleife, und damit beide Threads, abgebrochen und neu gestartet.

```
1  module ABRO:
2     input A, B, R;
3     output O;
4     loop
5        [
6        await A
7        ||
8        await B
9        ];
10       emit O
11    each R
12 end module
```

Listing 3.1: ABRO in Esterel

Das Verhalten des Programms hängt zur Laufzeit nicht davon ab, ob zuerst der eine oder der andere Thread abgearbeitet wird, wohl aber davon, wann das Signal `R` geprüft wird. Dies muss, da es sich um eine *Strong Abortion* handelt, vor der Ausführung der Threads passieren.

Ein Esterel-Compiler berücksichtigt diese Hierarchieabhängigkeit und legt die Ausführungsreihenfolge entsprechend fest. Können die Abhängigkeiten nicht aufgelöst werden, oder enthält das Programm einen Widerspruch oder eine Unentscheidbarkeit, wird es zurückgewiesen.

Wie man an Listing 3.1 sieht ist Esterel eine imperative Sprache. Neben imperativen existieren auch synchrone Datenflusssprachen, zum Beispiel die relationale Datenflusssprache *Lustre* [32] und die funktionale Datenflusssprache *Signal* [40]. Für diese Arbeit sind sie jedoch nicht von Bedeutung.

Synchrone Sprachen können auf vielfältige Weise übersetzt werden. Für Esterel existieren Compiler, die Kontrollflussgraphen, Ereignisgraphen, Logiknetzwerke oder endliche Automaten erzeugen [22]. Endliche Automaten erlauben es, synchrone Programme automatisiert auf Korrektheit zu prüfen. Die Erzeugung leidet jedoch unter

Zustandsexplosionen, die schon bei mittleren Programmgrößen die Übersetzung behindern können.

3.2. SyncCharts

Als grafische Modellierungssprache mit zu Esterel vollständig kompatibler mathematischer Semantik wurde SyncCharts entwickelt [5], auch als „Safe State Machines“ bekannt. Als synchrone Variante von *Statecharts* [34] nutzen sie deren Sprachelemente zum Ausdruck von Hierarchie, Preemption und Nebenläufigkeit. Im Gegensatz zu anderen Statecharts-Dialekten [8] bieten sie jedoch deterministische Nebenläufigkeit. Da SyncCharts eine *Hierarchie* von endlichen Automaten beschreiben, werden Zustandsexplosionen bei der Komposition von Systemen vermieden. In SyncCharts kann wie in Esterel auch auf die Abwesenheit von Signalen reagiert werden [61, Lecture 10].

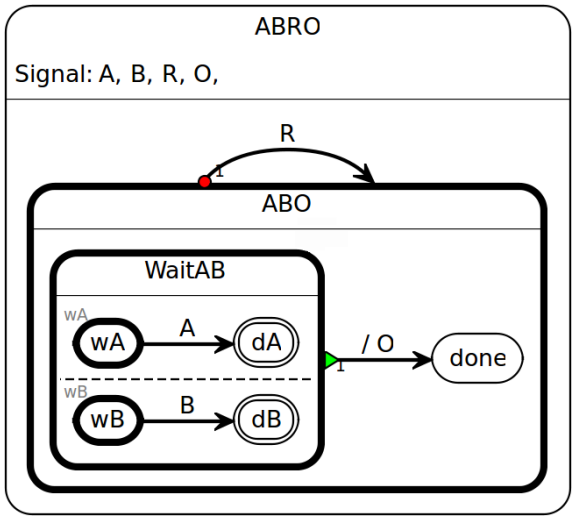


Abbildung 3.2.: ABRO als SyncChart

Abbildung 3.2 zeigt das Beispiel ABRO als SyncChart. Die beiden durch eine gestrichelte Linie getrennten Bereiche sind nebenläufig. Sie enthalten je einen Thread mit zwei Zuständen. Die Threads verharren in den initialen Zuständen `wA` und `wB` und nehmen die jeweilige Transition nach `dA` beziehungsweise `dB`, wenn das jeweils angezeichnete Signal anliegt. Die Zustände `dA` und `dB` sind finale Zustände – wenn sie erreicht werden, terminiert der sie umgebende Makrozustand `WaitAB`. Die mit `/O` beschriftete Transition sendet das Signal `O` aus. Das kleine grüne Dreieck bedeutet, dass es sich um eine *Normal Termination* handelt. Sie wird genommen, sobald alle Threads im Makrozustand `WaitAB` ihren Endzustand erreicht haben. Der den Makrozustand `WaitAB` und den Endzustand `done` enthaltende Makrozustand `ABO` wiederum kann durch ein Signal `R` sozusagen neu gestartet werden. Bei der entsprechenden Transition handelt es sich um eine Strong Abortion, die durch einen roten Kreis am

Anfang der Transition angezeigt wird. Ein Makrozustand kann auch durch eine *Weak Abortion* verlassen werden, das heißt, ohne Abbruch der im laufenden Tick stattfindenden Aktionen im Inneren. Entsprechende Transitionen haben keine besondere Markierung.

Zusätzlich zu den in Abbildung 3.2 gezeigten Elementen gibt es noch solche zum Ausdruck von *Supension*, das heißt zum Pausieren anderer Threads oder Zustände. Suspension wird in dieser Arbeit jedoch nicht betrachtet. Nicht in ABRO enthalten sind außerdem *Immediate-Transitionen*, die mit einem #-Symbol gekennzeichnet werden. Durch Immediate-Transitionen können in einem Tick mehrere Transitionen nacheinander genommen beziehungsweise im laufenden Tick erreichte Zustände sofort wieder verlassen werden.

3.3. Synchronous C (SC)

Synchronous C (SC), auch *SyncCharts in C* [62], ist eine makro-basierte Erweiterung der Programmiersprache C um Signale und Operatoren, die leichtgewichtiges Multithreading, Preemption und die Umsetzung hierarchischer Abhängigkeiten ermöglichen.

Die SC-Operatoren sind als Makros in C-Header-Dateien definiert, daher ist SC sehr portabel. SC-Programme können mit jedem C-Compiler, der einen Präprozessor mit Makroexpansion beinhaltet, kompiliert werden. Mit Hilfe der zusätzlichen Operatoren können in SC deterministische nebenläufige Programme geschrieben werden. SC eignet sich aber vor allem auch als Zielsprache zur Synthese aus SyncCharts, da es alle Sprachelemente von SyncCharts unterstützt.

Mit den SC-Operatoren `EMIT` und `PRESENT` können Signale gesendet und geprüft werden. Signale werden über eigene Datenstrukturen verwaltet und stehen global zur Verfügung. Da SC-Programme sequentiell abgearbeitet werden, ist ein Signal allerdings erst nach einem `EMIT` PRESENT, könnte also vorher mit dem Ergebnis ABSENT geprüft worden sein. Abschaltbare Prüfmechanismen können solche Fehler erkennen. Da das Verhalten eines SC-Programms von der Reihenfolge der Operatoren abhängt, kann keine perfekte Synchronizität angenommen werden, SC erfüllt also selbst die synchrone Hypothese nicht. SC-Programme können aber aus synchronen Sprachen synthetisiert werden und deren Verhalten korrekt abbilden.

SC ermöglicht Multithreading durch Koroutinen[2], verwendet also Rücksprungadressen, die vom *aufrufenden* Thread gesetzt werden. Die Kontextwechsel erfolgen durch `goto`-Sprünge. Da in C kein direkter Zugriff auf den Program Counter möglich ist, werden *Continuation Labels* verwendet. Diese werden durch Makros dort erzeugt, wo ein Verlassen beziehungsweise Wiedereintreten in den Ablauf eines Threads möglich ist.

Da bei der Makroexpansion komplette Codeblöcke eingefügt werden, ist es auch möglich, Code mit Switch-Case-Logik zu erzeugen.

[2] `http://en.wikipedia.org/w/index.php?title=Coroutine&oldid=385602930`

Zum Wechsel zwischen den Threads findet ein Sprung in den *Dispatcher* statt, der anhand von eindeutigen Prioritäten entscheidet, welcher Thread als nächstes ausgeführt wird. Die Logik von SC orientiert sich hier stark an der des Kiel Esterel Processor [42], dessen Scheduling-Mechanismus SC übernommen hat. Die Multithreading-Funktionalität in SC ist deterministisch, weil die Ausführungsreihenfolge durch die Prioritäten vorgegeben ist. SC setzt allerdings auf kooperatives Scheduling – ein Thread läuft so lange, bis er einen SC-Operator ausführt, der die Hoheit über den Prozessor wieder abgibt.

Da SC seine eigenen Funktionen zur Verwaltung und Steuerung der Threads mitbringt, können SC-Programme auf dem *„bare metal“*, also direkt auf der Hardware, ausgeführt werden. So entfällt der Overhead durch Betriebssysteme, die die Ausführung nichtdeterministisch und damit unvorhersehbar machen [4]. Der Verzicht auf Stack-Operationen beim Kontextwechsel macht SC darüber hinaus besonders leistungsfähig.

Die Thread-Verwaltung nutzt schlanke Datenstrukturen, allesamt Arrays, die über die Prioritäten indiziert werden. Die Prioritäten sind somit identisch mit den Process IDs (PIDs) und werden im Code auch als solche bezeichnet.

SC-Programme können beliebigen C-Code enthalten, zum Beispiel Variablen und Funktionsaufrufe. Allerdings ist es nicht möglich, Funktionen durch `goto`-Sprünge in ihrem Ablauf zu unterbrechen. Viele SC-Operatoren können daher nicht in Funktionen verwendet werden. Die Threads haben keine getrennten Speicherbereiche. Variablen sind also ebenso wie Signale global.

Mit dem Operator `FORK` erzeugte Threads werden als Child-Threads gespeichert und gegebenenfalls entsprechend behandelt, also zum Beispiel mit abgebrochen. Hierarchien bleiben so erhalten. Mit den Operatoren `SUSPEND` und `ABORT` prüfen Threads, ob eine Bedingung zum Pausieren oder zum Abbruch gegeben ist.

Listing 3.2 zeigt das Programm ABRO in SC. Auffällig sind die Prioritäten, die in `TICKSTART` in Zeile 1 und in den `FORK`s in den Zeilen 4, 8 und 9 angegeben sind. In `TICKSTART` wird die Priorität, mit der der Haupt-Thread auszuführen ist, bestimmt, in `FORK` die der erzeugten Child-Threads. Die Labels in den Zeilen 3, 7, 12, 16, 20 und 25 markieren jeweils den Beginn eines Zustands. Zustände werden nach diesen Labels benannt. Der erste Parameter in `FORK` gibt den Startzustand des neuen Threads an.

Da `ABOMain` immer die höchste Priorität im Programm besitzt, wird in jedem Tick zuerst mit `AWAIT(R)` geprüft, ob das Abbruchsignal `R` PRESENT ist. Ist dies der Fall, werden mit `TRANS` alle erzeugten Threads abgebrochen und die Ausführung bei `ABO` neu gestartet.

Abbildung 3.3 zeigt einen Ablaufplan des ersten Ticks mit Sprüngen in den Dispatcher. `TICKSTART` setzt im ersten Tick direkt mit dem darunterstehenden Code fort. Die Ausführung des eigentlichen Programms beginnt dann bei `ABO`. Hier wird der Thread `AB` erzeugt und danach mit `FORKE` in den Dispatcher gesprungen, der aber, da 4 immer noch die höchste Priorität ist, direkt zum angegebenen Continuation Label `ABOMain` springt. Die Zahlen in der orangenen Box, die den Dispatcher symbolisiert, geben an, welche PID der Dispatcher aufruft.

```
1  TICKSTART(4);
2
3  ABO:
4     FORK(AB, 1);
5     FORKE(ABOMain);
6
7  AB:
8     FORK(WaitA, 2);
9     FORK(WaitB, 3);
10    FORKE(ABMain);
11
12 WaitA:
13    AWAIT(A);
14    TERM;
15
16 WaitB:
17    AWAIT(B);
18    TERM;
19
20 ABMain:
21    JOIN;
22    EMIT(O);
23    TERM;
24
25 ABOMain:
26    AWAIT(R);
27    TRANS(ABO);
28
29 TICKEND;
```

Listing 3.2: ABRO in SC

Alle `AWAIT`-Statements beinhalten ein initiales `PAUSE`. Im ersten Tick beendet `AWAIT(R)` also die Ausführung des Threads mit der PID 4 direkt wieder. In `AB` werden dann die Threads `WaitA` und `WaitB` abgespalten und angesprungen. Sie werden wiederum über das initiale `PAUSE` für den Tick beendet. Der verbleibende Thread führt in `ABMain` `JOIN` aus, welches sich ebenfalls als `PAUSE` auswirkt, weil die Child-Threads noch nicht beendet sind. Die Ausführung endet mit der PID 0, die zur Beendigung der Tick-Funktion führt.

Hier sieht man zwei für diese Arbeit wichtige Eigenschaften von SC: Zum einen haben die Threads `WaitA` und `WaitB`, obwohl sie nebenläufig sind, unterschiedliche Prioritäten. Dies ist der Notwendigkeit von eindeutigen PIDs zur internen Verwaltung geschuldet, dient aber auch dem deterministischen Scheduling. Zum anderen können durch `FORK` Threads mit höherer Priorität erzeugt werden, die dann vor dem Parent-Thread ausgeführt werden. Die Folge der Prioritäten ist also nicht monoton fallend. Darüber hinaus können Threads auch mit `PRIO` ihre eigenen Priorität ändern. Ein solcher Wechsel ist in manchen Programmen zur sequentiellen Umsetzung wechselnder Abhängigkeiten innerhalb eines Ticks nötig.

Wie der Ausdruck `AWAIT(A)` in Zeile 13 des Beispiels 3.2 nach der Expansion durch den Präprozessor aussieht, ist in Listing 3.4(a) gezeigt. Dort sind Ausgaben zur Verfolgung des Programmablaufs enthalten. Ein umgebendes `do-while`-Konstrukt sorgt dafür, dass ein Makro auch als einzelner, ungeklammerter Ausdruck nach einem if-

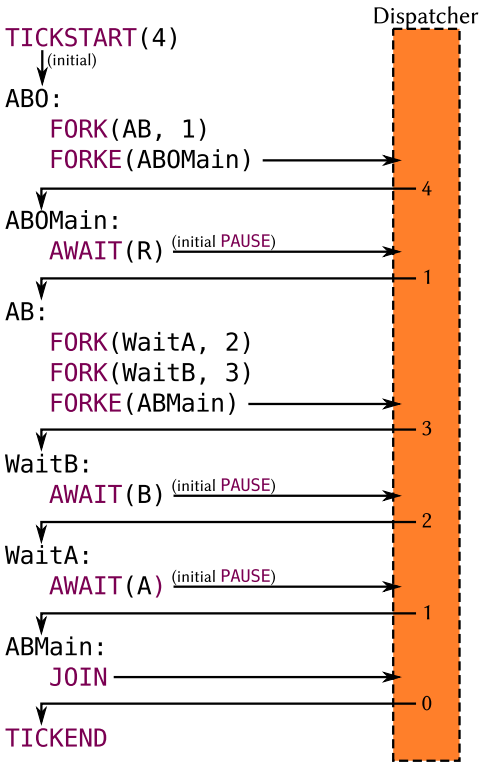


Abbildung 3.3.: Ablauf des ersten Ticks in ABRO in Synchronous C

Statement genutzt werden kann. Listing 3.4(b) enthält eine gekürzte Fassung. In der ersten Zeile wird zunächst zum automatisch generierten Label `_LL195` gesprungen, was dem Verhalten bei Abwesenheit des Signals entspricht, also einem `PAUSE`. In späteren Ticks wird die Ausführung jedoch beim Label `_L195` in Zeile 2 fortgesetzt, denn vor dem Sprung in den Dispatcher wird dieses in Zeile 9 als Continuation Label festgelegt. Das als Array-Index verwendete `_cid` ist die PID des aktuellen Threads. Zeile 10 bewirkt den Sprung in den Dispatcher.

Für den Fall, dass das geprüfte Signal `A` ABSENT ist, wird die Tatsache, dass es bereits geprüft wurde, in Zeile 4 notiert. Sollte `A` später im selben Tick emittiert werden, würde dies als Fehler erkannt. Wenn `A` PRESENT ist, wird die Ausführung direkt nach dem Konstrukt fortgesetzt.

Beide Listings wurden mit der für diese Arbeit modifizierten Version der Header-Files erzeugt, die Unterschiede zum originalen SC sind jedoch marginal.

In den Tabellen 3.1(a) und 3.1(b) sind die wichtigsten SC-Operatoren mit einer kurzen Erklärung aufgelistet. Mit * markierte Operatoren springen in den Dispatcher und können einen Thread-Wechsel auslösen. Mit † markierte Operatoren erzeugen ein Continuation Label als Sprungziel.

```
do {
   tickInstrCnt++;
   printf("%-9s %d/%s ", "AWAIT:", _cid, state[_cid]);
   printf("initial pause\n");
   goto _LL195;
   _L195:
   _checkSIG_ID(A, 4);
   if (!signals[A]) (_presence_tested[A]=1);
   if (signals[A]) {
      tickInstrCnt++;
      printf("%-9s %d/%s ", "AWAIT:",
            _cid, state[_cid]);
      printf("determines %s/%d present, proceeds\n",
            s2signame[A], A);
   }
   else {
      tickInstrCnt++;
      printf("%-9s %d/%s ", "AWAIT:",
            _cid, state[_cid]);
      printf("determines %s/%d absent, waits\n",
            s2signame[A], A);
      _LL195:
      _pc[_cid] = &&_L195;
      statePrev[_cid] = state[_cid];
      state[_cid] = "_L195";
      goto _L_PAUSEG;
   }
}
while (0);
```

(a) Vollständig

```
   goto _LL195;
   _L195:
   if (!signals[A])
      (_presence_tested[A]=1);
   if (signals[A]) {
   }
   else {
      _LL195:
      _pc[_cid] = &&_L195;
      goto _L_PAUSEG;
   }
```

(b) Gekürzt

Abbildung 3.4.: Expandierter Synchronous C-Operator `WAIT(A)`

Viele Operatoren sind Kurzformen für Konstrukte, die sich aus den Basis-Operatoren zusammensetzen lassen. Einige sind allerdings optimiert und sparen eventuell einen Sprung in den Dispatcher ein.

Mit `SIGNAL` wird ein lokales Signal deklariert. Da Signale in SC aber immer global sind, bewirkt `SIGNAL` bislang lediglich eine Rücksetzung auf ABSENT. Mit `CALL` und `RET` ist der einfache, nicht verschachtelte Aufruf von Subroutinen möglich.

Das bereits erklärte `FORK` von SC ähnelt etwas dem `par`-Statement von PRET-C. Dessen parallele Zweige werden aber in einer festen, durch die Reihenfolge im Code vorgegebenen Reihenfolge sequentiell abgearbeitet. Zudem prüft dort jeder Zweig mit einem sogenannten `checkabort`-Statement, ob Abbruchbedingungen vorliegen. SC setzt hingegen hierarchische Abhängigkeiten um: Abbruchbedingungen werden nur einmal pro Tick vom Parent-Thread geprüft.

3.4. Synthese von SC aus SyncCharts

Der SyncCharts-Compiler von Amende [3] erzeugt als Plugin im KIELER-Framework SC-Code direkt aus SyncCharts.

Zwar ist die Übersetzung von SyncCharts nach C über Esterel als Zwischensprache auf vielfältige Weise möglich. Der Kontrollfluss muss dabei aber in Schleifenkonstrukte umgewandelt werden. Zustandstransitionen sind nicht mehr als solche erkennbar. Der Bezug zum SyncChart geht im Esterel-Programm verloren. Mit einem weiteren Compiler kann ein Esterel-Programm dann über ein Zwischenformat in C-Code übersetzt werden. Auch die in Kapitel 2 genannten Methoden zur Verteilung synchroner Programme verarbeiten den aus Esterel generierten Zwischencode. Der resultierende C-Code ist zwar effizient, zeigt aber keinerlei Ähnlichkeiten mit dem SyncChart mehr.

Der SyncCharts-Compiler spart hingegen mehrere Übersetzungsschritte ein. Er übersetzt nebenläufiges Design direkt in Threads, deren Zustände denen im SyncChart entsprechen. Zusätzliche Threads werden eventuell zur Abbildung von Hierarchien generiert, eine Zustandsexplosion gibt es aber nicht.

Die Funktionsweise des Compilers verdeutlicht den engen Zusammenhang zwischen SC-Code und SyncChart und soll hier vereinfacht erklärt werden.

In einem SyncChart wie dem in Abbildung 3.2 gezeigten gibt es verschiedene Abhängigkeiten:

Kontrollflussabhängigkeiten sind solche Abhängigkeiten, die zwischen durch eine Transition verbundenen Zuständen bestehen. Zielzustände sind offensichtlich von Startzuständen abhängig.

Hierarchieabhängigkeiten bestehen zwischen inneren Zuständen und jeweils den äußeren Makrozuständen, die sie enthalten. Allerdings wird unterschieden, ob eine Strong Abortion möglich ist, denn nur dann sind die inneren Zustände vom äußeren abhängig, andernfalls ist es umgekehrt.

Signalabhängigkeiten sind leicht erkennbar die Abhängigkeiten zwischen Sendern und Empfängern je eines Signals, falls diese gleichzeitig ausgeführt werden können.

Transitionsabhängigkeiten existieren zwischen ausgehenden Transitionen eines Zustands, die im SyncChart mit Prioritäten in eine eindeutige Rangordnung gebracht werden.

Der Compiler betrachtet Signalabhängigkeiten genaugenommen als Abhängigkeiten zwischen Tupeln aus Zustand und ausgehender Transition, zur Vereinfachung wird dies hier aber nicht berücksichtigt. Mehrfache und gegenseitige Signalabhängigkeiten beherrscht er ebenfalls. Für diese Arbeit genügt es jedoch zu wissen, dass an passender Stelle im generierten SC-Code `PRIO`-Statements eingefügt werden.

Abbildung 3.5 zeigt eine modifizierte Version von ABRO als SyncChart. Dieses ABRO wurde gegenüber dem in Abbildung 3.2 um das Aussenden des Signals `B`, falls `A` PRESENT ist, erweitert. Die Zustände sind alle benannt, um eine Unterscheidung zu ermöglichen.

Wie man leicht sehen kann, bestehen

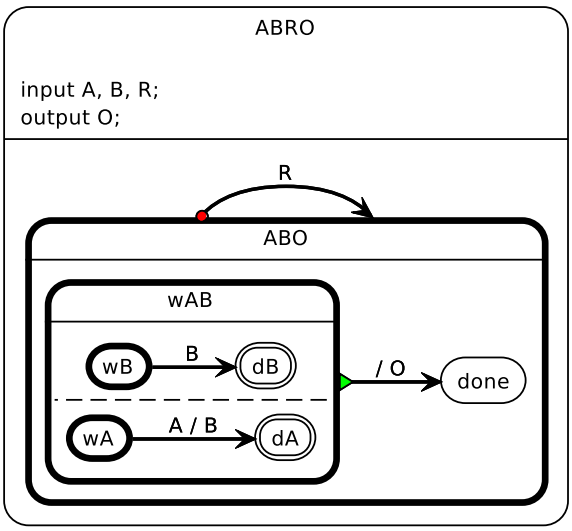


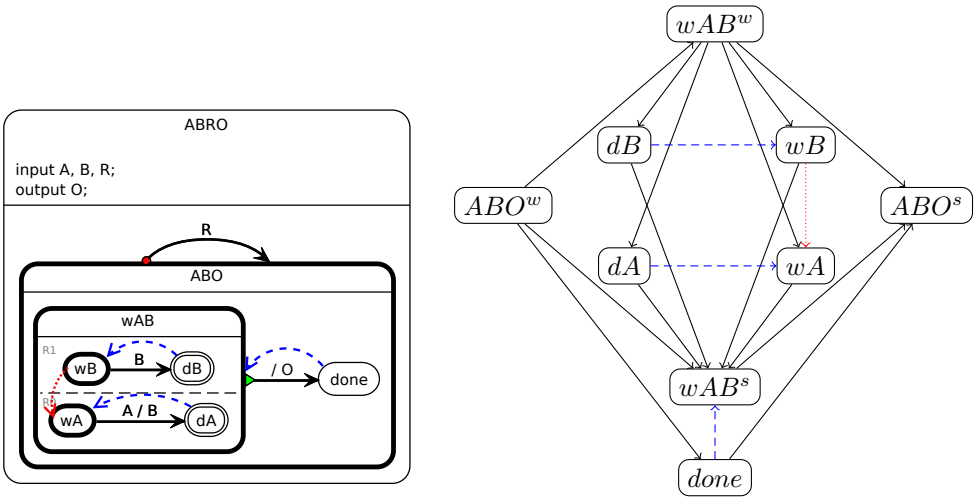
Abbildung 3.5.: Modifiziertes ABRO mit Signalabhängigkeit (aus: Amende [3])

- zwischen dem Zustand `wB` und `wA` eine Signalabhängigkeit,
- zwischen jeweils `dB` und `wB`, `dA` und `wA` sowie `done` und `wAB` Kontrollflussabhängigkeiten sowie
- zwischen `done` und `ABO`, `wAB` und `ABO`, `dA` und `wAB`, `wA` und `wAB`, `dB` und `wAB`, `wB` und `wAB` Hierarchieabhängigkeiten.

Die Hierarchieabhängigkeiten vom Makrozustand `ABO` müssen zusätzlich unterschieden werden. Die Transition in den Zustand `done` ist eine Normal Termination, `ABO` ist also abhängig von `wAB`. Da eine Strong Abortion möglich ist, existiert aber auch eine umgekehrte Abhängigkeit. Der Compiler unterscheidet daher zwischen *starken* und *schwachen* Teilen eines hierarchischen Zustands. Die starken Teile sind die, die Abbruchbedingungen für Strong Abortions prüfen, alle anderen sind schwach. Durch diese Unterscheidung kann Code generiert werden, der zunächst die „starken“ Teile eines Zustands ausführt, und nach dessen Child-Threads die „schwachen“ Teile.

Der Compiler erzeugt mit diesen Informationen aus einem SyncChart einen Abhängigkeitsbaum. Dieser muss gerichtet und azyklisch sein, zirkuläre Abhängigkeiten können nur durch Fehler im SyncChart entstehen. Abbildung 3.6(a) zeigt den ABRO-SyncChart mit eingezeichneten Signal- und Kontrollflussabhängigkeiten. Die einzige Signalabhängigkeit ist rot gepunktet, die Kontrollflussabhängigkeiten sind blau gestrichelt gezeichnet. In Abbildung 3.6(b) ist der zugehörige Abhängigkeitsbaum vereinfacht gezeigt.

Um die kausalen Abhängigkeiten im SyncChart in eine zeitliche Folge umzuwandeln, ordnet der Compiler den generierten Abhängigkeitsbaum topologisch. Dann weist er den Zuständen entsprechend ihrer Position in der Ordnung Prioritäten zu.



(a) SyncChart mit Signal- und Kontrollflussabhängigkeiten

(b) Abhängigkeitsbaum

Abbildung 3.6.: Abhängigkeitsbaum und modifiziertes ABRO mit Signal- und Kontrollflussabhängigkeiten (nach: Amende [3])

Die topologische Ordnung wird generiert, indem sukzessive ein Wurzelknoten gesucht und dem Baum entnommen wird. Wurzelknoten sind nur von anderen abhängig, haben also keine Nachfolger. Sie müssen folglich zuletzt ausgeführt werden, bekommen daher die niedrigste Stelle in der Ordnung und somit die niedrigste Priorität. Wird ein Wurzelknoten entfernt, so entstehen ein oder mehrere neue Wurzelknoten, die ebenfalls nacheinander entnommen und eingeordnet werden, bis der Baum vollständig in die Ordnung übertragen wurde.

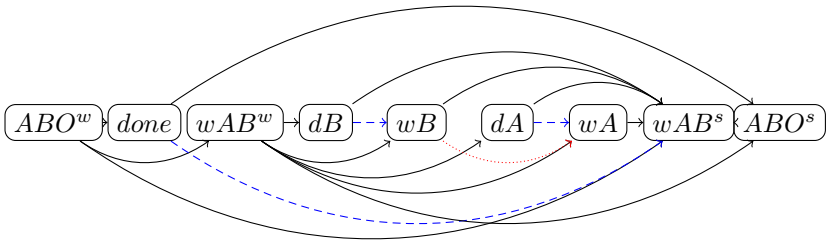


Abbildung 3.7.: Topologisch geordneter Abhängigkeitsbaum (nach: Amende [3])

In Abbildung 3.7 ist der Abhängigkeitsbaum aus Abbildung 3.6(b) topologisch sortiert dargestellt. Die Prioritäten der einzelnen Zustände bestimmen sich nach der Position, im Beispiel von links nach rechts aufsteigend. Die Prioritäten zur Initialisierung der Threads werden den jeweiligen Startzuständen entnommen.

Auch ohne Abhängigkeit zwischen parallelen Regionen wie beim modifizierten ABRO werden unterschiedliche Prioritäten vergeben. Wie in Abschnitt 3.3 erklärt,

müssen diese eindeutig sein. Welcher Thread die höhere Priorität bekommt, ist ohne Abhängigkeiten jedoch nicht vorhersagbar. Das Wissen über ihre Nebenläufigkeit geht also bei der Codeerzeugung verloren.

Nachdem die Prioritäten ermittelt wurden, wird der konkrete SC-Code mit Hilfe von Templates erzeugt.

Operator	Funktion
`SIGNAL(S)`	Initialisiert ein lokales Signal `S`.
`EMIT(S)`	Emittiert Signal `S`.
`SUSTAIN(S)`*†	Hält Signal `S` PRESENT. Kurz für `l: EMIT(S); PAUSE; GOTO(l)`.
`PRESENT(S)`	Liefert 1, wenn `S` PRESENT ist, 0 andernfalls.
`PRESENTELSE(S,` l_{else}`)`	Springt zu l_{else}, wenn `S` ABSENT ist, setzt sonst fort. Kurz für: `if (!PRESENT(s)) GOTO(`l_{else}`)`.
`PRESENTEMIT(S, T)`	Falls `S` PRESENT ist, emittiere `T`. Kurz für `if (PRESENT(S)) EMIT(T)`.
`AWAIT(S)`*†	Warte auf Signal `S`. Kurz für l_{else}`: PAUSE; PRESENT(s,` l_{else}`)`.
`AWAITI(S)`*†	Immediate-Variante von `AWAIT(S`. Kurz für `GOTO(l);` l_{else}`: PAUSE; l: PRESENT(s,` l_{else}`)`.
`PRESENTPRE(S,` l_{else}`)`	Liefert 1, falls `S` im vorigen Tick PRESENT war, andernfalls 0.
`PRESENTPREELSE(S,` l_{else}`)`	Springt zu l_{else} falls `S` im vorigen Tick PRESENT war, setzt sonst fort. Kurz für `if (!PRESENTPRE(S)) GOTO(`l_{else}`)`.
`GOTO(l)`	Springt zu Label `l`.
`CALL(l)`	Ruft Funktion `l` auf.
`RET`	Kehrt vom Funktionsaufruf zurück.

(a) Signal-Operatoren und sequentielle Kontrolloperatoren

Operator	Funktion
`TICKSTART(p)`*	Startet den Tick, weist dem Haupt-Thread die Priorität `p` zu.
`TICKEND`	Beendet den Tick, gibt 1 zurück, falls es noch aktive Threads gibt.
`PAUSE`*†	Deaktiviert aktuellen Thread für den laufenden Tick.
`PAUSEG`*`(l)`	Kurz für `PAUSE; GOTO(l)`.
`HALT`*†	Kurz für `l: PAUSE; GOTO(l)`.
`TERM`*	Beendet den aktuellen Thread.
`ABORT`	Bricht vom aktuellen Thread erzeugte Threads ab.
`TRANS(l)`	Kurz für `ABORT; GOTO(l)`.
`SUSPEND`*`(cond)`	Suspendiert (pausiert) aktuellen Thread und vom aktuellen Thread erzeugte Threads, falls `cond` wahr ist.
`FORK(l, p)`	Erzeugt einen neuen Thread mit Startadresse `l` und Priorität `p`.
`FORKE(l)`*	Beendet `FORK`-Vorgang und setzt bei `l` fort. Muss `FORK`s folgen.
`JOIN`*†	Wartet bis vom aktuellen Thread erzeugte Threads terminiert sind. Kurz für l_{else}`: JOINELSE(`l_{else}`)`.
`JOINELSE`*†`(`l_{else}`)`	Fährt fort, falls vom aktuellen Thread erzeugte Threads terminiert sind, springt andernfalls zu l_{else}.
`PRIO`*†`(p)`	Setzt die Priorität des aktuellen Threads auf `p`.

(b) Thread-Operatoren

Tabelle 3.1.: Operatoren in Synchronous C (nach: von Hanxleden [62])

4. Multicore-Plattformen

Im Folgenden werden Multicoresysteme in Abgrenzung zu anderen Techniken und die Probleme moderner Prozessoren in Bezug auf vorhersagbares Verhalten erläutert. Nach Vorstellung der Kriterien für die Auswahl werden mehrere Plattformen aus verschiedenen Bereichen betrachtet. Am Ende des Kapitels wird eine Auswahl getroffen und begründet.

4.1. Einleitung

Ein Multicore-Prozessor ist ein Chip, der mehrere unabhängige Prozessorkerne beherbergt, die über ein Bussystem verbunden sind und in der Regel auf einen gemeinsamen Speicher zugreifen. Je nach Ausführung werden verschiedene Cache-Ebenen geteilt. Bis auf Bus und Cache hat also jeder Kern alles, was er zur Ausführung eines Programms benötigt. Sind die Kerne identisch, spricht man von Symmetric Multiprocessing (SMP), andernfalls von asymmetrischem Multiprocessing. Bei asymmetrischen Multicore-Prozessoren sind die Kerne auf unterschiedliche Aufgaben ausgerichtet, haben entsprechend unterschiedliche Funktionen und Befehlssätze.

Asymmetrische Multicore-Prozessoren könnten in der Zukunft aufgrund ihrer Energieeffizienz an Bedeutung gewinnen. Woo und Lee [65] haben Amdahls Gesetz [2] mit Betrachtungen zur Energieeffizienz kombiniert und sind zu dem Schluss gekommen, dass ein SMP-Prozessor mit zunehmender Zahl von Kernen seine Energieeffizienz komplett einbüßen kann. Sie empfehlen asymmetrische Systeme mit kleinen effizienten Kernen für die parallelisierbaren Aufgaben und einem vollausgestatteten Kern für die sequentiellen Teile des Programms.

Von Multithreading spricht man, wenn ein Prozessor die Ausführung mehrerer Threads unterstützt, aber nur Teile seines Kerns mehrfach besitzt, zum Beispiel Registersätze oder die arithmetisch-logische Einheit (ALU). Er hat also die Fähigkeit, seine Komponenten besser auszulasten, indem er die Befehle eines Threads abarbeitet, wenn kein anderer dieselbe Komponente benötigt. Beispielsweise kann ein Thread durch einen Speicherzugriff verzögert werden, weil der Hauptspeicher langsamer ist als die CPU. Bis zum Eintreffen der Daten kann ein anderer Thread aber Rechenoperationen durchführen. Der Prozessor ist in der Lage, den *Ausführungskontext* zu wechseln [37]. Simultaneous Multithreading (SMT) ist eine Weiterentwicklung der Block-Interleaving-Technik, bei der die CPU einen Thread-Wechsel erst bei Wartezeiten vornimmt. Beim SMT existieren für die Threads eigene Instruktions-Pipelines,

der Prozessor führt die Instruktionen nach Verfügbarkeit der Komponenten aus, ohne erst Blockaden erkennen zu müssen[1].

Prozessoren wie der in Kapitel 2 erwähnte KEP und der ARPRET [4] unterstützen Software-Multithreading durch Hardware-Erweiterungen. Sie weisen aber nicht jedem Thread gleich viel Rechenzeit zu, sondern wenden zum Beispiel einen prioritätenbasierten Scheduling-Mechanismus an. Der Kontextwechsel wird also von der Software gesteuert, es handelt sich nicht um SMT.

Abbildung 4.1(a) zeigt schematisch den Aufbau einer Singlecore-CPU. Es gibt nur eine Instruktions-Pipeline, symbolisiert durch vier Kästchen. Dem einzigen Thread stehen Fließkommaeinheit (FPU), arithmetisch-logische Einheit (ALU) und weitere Komponenten exklusiv zur Verfügung. Der Multithreading-fähige Prozessor in Abbildung 4.1(b) hat zwei Pipelines und führt Instruktionen je nach Verfügbarkeit der Komponenten aus. Beim Mehrkernsystem in Abbildung 4.1(c) ist der Prozessorkern doppelt vorhanden. Jeder Kern hat seinen eigenen Level-1-Cache, die beiden konkurrieren aber aber um Bus, Level-2-Cache und Zugriff auf den Hauptspeicher.

Zu unterscheiden sind SMP- und SMT-Prozessoren von Multiprozessorsystemen. Dabei handelt es sich um Systeme, die mehrere Prozessoren beinhalten, die aber nicht unbedingt auf einem Chip untergebracht und entsprechend auch nicht so eng gekoppelt sind.

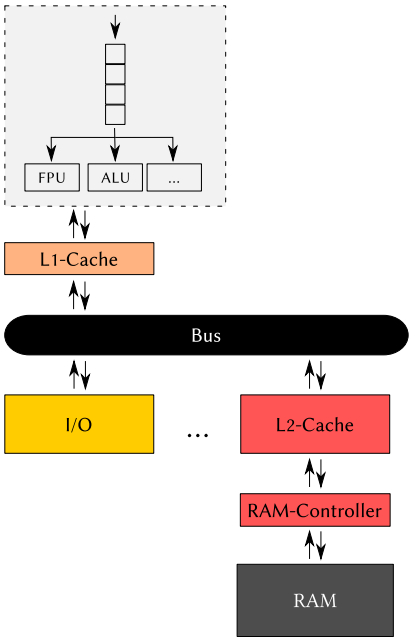
Multiprozessorsysteme können als Non-Uniform Memory Architecture (NUMA) ausgeführt sein, das heißt, die Speicherzugriffszeit hängt von der Position des Prozessors im System ab. Zu unterscheiden sind sie auch von SIMD-Systemen wie Vektorprozessoren, die eine Operation simultan auf einer großen Zahl von Datensätzen ausführen.

Obwohl die „Multicore-Revolution“ [35] bereits seit einigen Jahren für Veränderungen im Desktop-Bereich sorgt, weil Leistungssteigerungen durch Erhöhung des Takts nur noch begrenzt und unter zu hohem Energieaufwand möglich sind [30], sind symmetrische Multicore-Systeme im Bereich eingebetteter Systeme noch selten. Oft handelt es sich um verteilte Echtzeitsysteme, bei denen Singlecore-CPUs nah an Sensoren und Aktuatoren eingesetzt und über ein Netzwerk verbunden werden (VL distributed). Asymmetrische Systeme finden sich in Form eines Hauptprozessors mit zusätzlichem DSP. Sie sind optimiert für Kommunikationsanwendungen wie zum Beispiel Mobiltelefone, die die Verarbeitung des Audio-Datenstroms an den DSP-Kern auslagern.

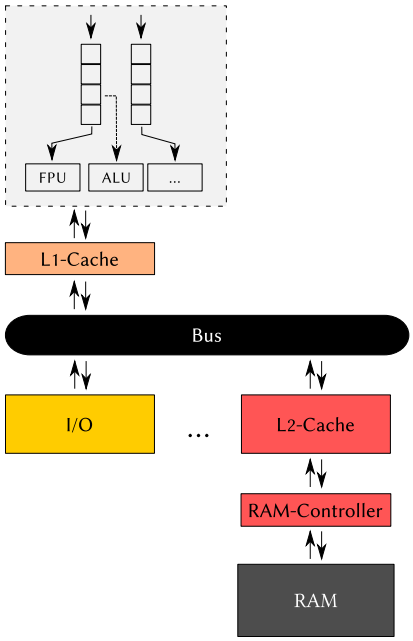
Die Ausstattungsmerkmale von Multicore- und Multithreading-CPUs, die für eingebettete Systeme vorgesehen sind, unterscheiden sich erheblich. Manche besitzen zusätzlich oder ausschließlich Speicher, die dem jeweiligen Kern zur direkten und ausschließlichen Nutzung zur Verfügung stehen, in Mehrkernprozessoren also schnell und kollisionsfrei genutzt werden können. Solche Speicher werden gelegentlich als *Scratchpad*-RAM bezeichnet[2] und sind meistens als schneller Static Random Access

[1] http://www.tecchannel.de/server/prozessoren/402289/cpu_grundlagen_multithreading/index7.html

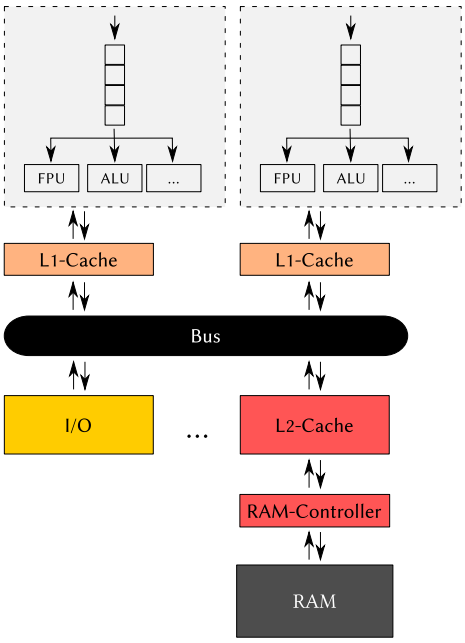
[2] http://en.wikipedia.org/w/index.php?title=Scratchpad_RAM&oldid=385639520



(a) Singlecore



(b) Simultaneous Multithreading (SMT)



(c) Symmetric Multiprocessing (SMP)

Abbildung 4.1.: Prozessorarten (vereinfacht)

Memory (SRAM) ausgeführt. Unterschiedlich sind auch die Kommunikationsmechanismen. Manche Prozessoren verfügen über Message-Passing-Funktionalitäten, bei anderen können die Kerne nur über den gemeinsamen Speicher („*Shared Memory*“) Informationen austauschen. Seltener sind Hardware-Synchronizer.

Prozessoren für eingebettete Systeme sind oft als System on a Chip (SoC) beziehungsweise Multiprocessor System on a Chip (MPSoC) ausgeführt, enthalten also auch noch Hauptspeicher, Bus-Controller oder weitere Komponenten. SoCs sind kostengünstiger herzustellen, aber auch weniger flexibel. Sie sind stark auf Kommunikation mit der Umgebung ausgelegt und haben viele Schnittstellen. Besitzt der Chip aber keinen Speichercontroller für externe Speicher, so ist eine Erweiterung nur schwer möglich.

4.2. Indeterminismen

Während es mit synchronen Sprachen möglich ist, das logische Verhalten eines reaktiven Systems perfekt zu verifizieren und vorherzusagen, erfordern Vorhersagen über das zeitliche Verhalten Wissen über die Hardware-Plattform, insbesondere über den Prozessor.

Moderne Prozessoren haben Interrupt-Controller, die es erlauben, Interrupt Requests umzulenken oder komplett zu sperren. Sie sind in vielerlei Hinsicht optimiert, nur nicht auf vorhersagbares Verhalten. In mehrstufigen Pipelines können Instruktionen sich gegenseitig beeinflussen, umsortiert, spekulativ ausgeführt oder parallelisiert werden und somit Konflikte auslösen[3]. Sprungvorhersagen können sich als falsch erweisen. Der Hauptspeicher ist langsamer als die CPU, daher kann es bei Zugriffen zu Verzögerungen kommen, wenn die Daten nicht im Cache liegen oder abgelegt werden können. Cache-Inhalte können aber invalidiert werden. Caches unterschiedlicher Größe und Organisation können über Kohärenz-Einheiten verbunden sein, und ihre Zugriffszeiten unterscheiden sich erheblich. DRAMs benötigen Refresh-Zyklen, während derer kein Speicherzugriff möglich ist. Peripherie kann den Hauptspeicher durch DMA-Zugriffe blockieren.

Alle diese Optimierungen verbessern die mittlere Ausführungszeit, Aussagen über die tatsächliche oder die maximale Ausführungszeit (Worst Case Execution Time, WCET) sind jedoch unzuverlässig. Im schlimmsten Fall ist die maximale Ausführungszeit sogar unbegrenzt, weil die Ausführung unfair ist oder eine Komponente des Systems eine Ressource blockiert. Der Bedarf an energieeffizienten Systemen hat darüber hinaus zu Erweiterungen geführt, die im Betrieb ungenutzte Teile von Prozessoren heruntertakten oder ganz abgeschalten. Je nachdem, in welchem Energiesparmodus sich die CPU oder ein Teil von ihr befindet, kann eine Operation schneller oder sehr viel langsamer als erwartet ausgeführt werden.

Moderne Prozessoren können also als geradezu chaotisch bezeichnet werden [43, 38], und alle diese Probleme zählen umso mehr, wenn mehrere Prozessorkerne Haupt-

[3] http://en.wikipedia.org/w/index.php?title=Hazard_%28computer_architecture%29&oldid=387882539

speicher, Peripherie, Caches oder einen Bus gemeinsam nutzen. Wie Lickly et al. [43] schreiben, ist die Kenntnis der WCET jedoch Grundlage jeder Implementierung eines Echtzeitsystems. Kann diese nur pessimistisch abgeschätzt werden, so muss das System, wenn Echtzeitkriterien garantiert erfüllt werden sollen, um mehrere Größenordnungen langsamer betrieben werden als im Mittel nötig.

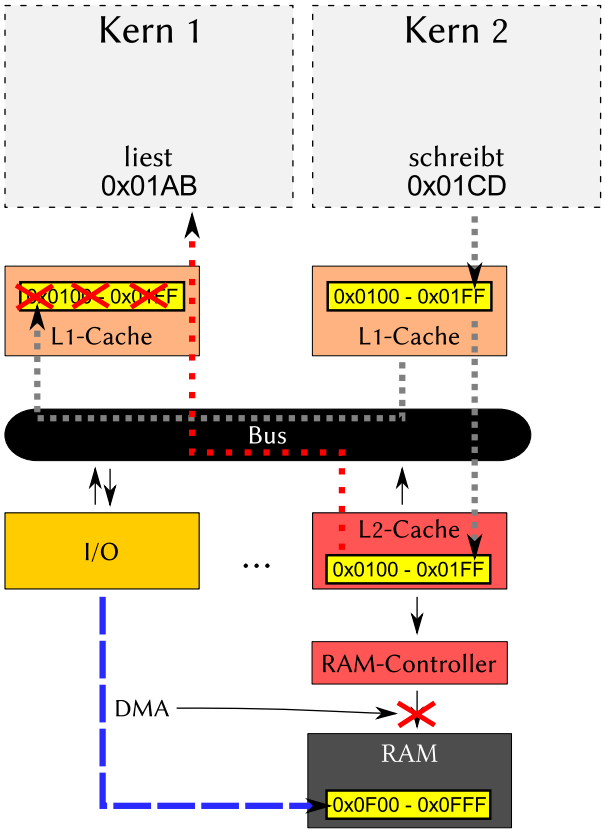


Abbildung 4.2.: Konkurrierende Speicherzugriffe

Abbildung 4.2 zeigt konkurrierende Speicherzugriffe auf einem Dualcore-System. Sowohl der Level-2- als auch die beiden L1-Caches enthalten eine Kopie eines Speichersegments, im Beispiel 0x0100 - 0x01FF. Je nach Prozessor führt der Schreibvorgang von Kern 2 (grau, eng gepunktet) in eine Adresse dieses Bereichs dazu, dass der gesamte Bereich im Level-2-Cache von Kern 1 über den Bus invalidiert, also für veraltet erklärt wird. Ein Lesevorgang von einer anderen Adresse im selben Segment durch Kern 1 (rot, weit gepunktet) wird je nach Zeitpunkt aus dem Level-1- oder Level-2-Cache bedient, kann also unterschiedlich schnell vonstatten gehen. Greift dazu noch Peripherie (I/O) per Direct Memory Access (DMA) auf den Hauptspeicher zu (blau, lang gestrichelt), so können Lesezugriffe in nicht im Cache vorgehaltene Bereiche stark verzögert werden.

4.3. Kriterien zur Auswahl einer Plattform

Aufgrund der in Abschnitt 4.2 genannten Probleme soll im folgenden Vergleich möglicher Multicore-Plattformen für Echtzeit-Systeme – auch wenn letztendlich die Leistungssteigerung im Vordergrund steht – besonderes Augenmerk möglichen Indeterminismen gelten. Für ein reaktives System zu bevorzugen sind SMP- und SMT-Prozessoren, wobei bei SMT beachtet werden muss, ob den Threads ein fester Anteil der zur Verfügung stehenden Rechenzeit zugewiesen wird. Hardware-Multithreading kann zu Schwankungen in der Ausführungszeit führen. Ist dies jedoch ausgeschlossen und findet ein Thread-Wechsel unabhängig vom Programmcode statt, so können SMT-Prozessoren wie echte Multicore-Systeme mit geringerer Gesamtleistung betrachtet werden.

Die Prozessorkerne sollen allerdings unter gleichen Voraussetzungen arbeiten. Die völlig unvorhersagbaren NUMA-Systeme werden daher nicht betrachtet, ebenso wenig asymmetrische Systeme mit DSPs. Die einem vollausgestatteten Kern zur Seite gestellten DSPs sind in der Regel zu spezialisiert, auf Datenfluss optimiert und nicht universell programmierbar. Weiterhin werden keine Desktop-Prozessoren betrachtet. Unter den handelsüblichen Desktop-Prozessoren gehören Mehrkernsysteme zwar mittlerweile zur Normalität, sie sind aber auch in Hinblick auf die mittlere Ausführungszeit am meisten hochgezüchtet und damit am schlechtesten vorhersagbar. Entsprechende Plattformen sind überdies mit viel Peripherie ausgestattet, die Einfluss auf die Abläufe nehmen kann. Grafikprozessoren (Graphics Processing Units, GPUs) scheiden, auch wenn sie universell programmierbar[4] sind, schon aufgrund ihres hohen Energiebedarfs [5] für die Entwicklung eingebetteter Echtzeitsysteme aus, zudem nutzen sie als SIMD-Systeme Datenparallelität statt der in dieser Arbeit angestrebten Ausführungsparallelität.

Eine Implementierung von Synchronisations- und Kommunikationsmechanismen hängt stark davon ab, welche Funktionen die Plattform zur Verfügung stellt. Direktes Message-Passing und ähnliche Funktionen sind der als fehlerträchtig geltenden Shared-Memory-Kommunikation [12, S. 90] vorzuziehen. Wie die Kerne miteinander kommunizieren können, hängt zudem von ihrer Anordnung ab; zum Beispiel gibt es neben Prozessoren, bei denen die Kerne über einen einfachen Bus verbunden sind, auch Ring- und Mesh-Systeme. Jeder Kern sollte mit jedem anderen kommunizieren können, damit Signale effizient global gesendet werden können. Konkurrierender Speicherzugriff lässt sich mit Scratchpad-RAMs verhindern, abschaltbare Caches könnten bei Shared-Memory-Kommunikation für ein nahezu gleichbleibendes Zeitverhalten sorgen. Beides könnte für zukünftige Projekte nützlich sein. Lock-Step-Debugging ist bei der Entwicklung von Vorteil, da die Anzeigemöglichkeiten stark eingeschränkt sind. Für wissenschaftliche Arbeiten ist außerdem die freie Zugänglichkeit von Quellcode und Entwicklungswerkzeugen, aber auch die leichte Programmierbarkeit des Prozessors von Belang. SCmc soll „bare metal“, also direkt auf

[4] General-Purpose computation on Graphics Processing Units (GPGPU) `http://gpgpu.org/about`

[5] Zum Beispiel benötigt der nVidia Tesla C1060 bis zu 188W, siehe `http://www.nvidia.com/object/product_tesla_c1060_us.html`.

dem Prozessor laufen, ohne Virtualisierungsschichten oder ein Echtzeitbetriebssystem (RTOS) dazwischen. Der Prozessor soll außerdem mehr als zwei Kerne haben.

4.4. CPUs

Zunächst werden einige Prozessoren betrachtet, die einzeln erhältlich sind und daher ein Motherboard benötigen – oder Lötarbeiten erfordern.

4.4.1. Freescale QorIQ P4080

Die Prozessoren der QorIQ-P4-Serie von Freescale, ehemals Motorola Semiconductor, haben Power-Architektur-Kerne[6], der P4080 ihrer acht[7]. Mit dieser Rechenleistung und drei Cache-Ebenen scheint er eher mit Desktop-Prozessoren vergleichbar. Allerdings ist er mit zahlreichen Schnittstellen ausgestattet und wird im militärischen Bereich durchaus in eingebetteten Systemen genutzt[8]. QorIQ-Prozessoren verfügen über Message-Passing-Befehle zur effizienten Kommunikation der Kerne untereinander. Prozessoren dieser Größenordnung sind in der Regel ohne Betriebssystem nicht komfortabel zu programmieren. Der P4080 ist außerdem eines der teuersten Systeme am Markt[9].

4.4.2. IBM/Sony/Toshiba Cell

Die Prozessoren der von IBM, Sony und Toshiba gemeinsam entwickelten Cell-Serie [50] sind asymmetrische Multicore-Prozessoren, das heißt, die Kerne unterscheiden sich in Leistung, Anbindung an die Peripherie und Funktionsumfang. Einem 64-Bit-PowerPC-Kern, genannt PowerPC Processing Element (PPE), stehen acht Vektorrecheneinheiten mit 128 Registern zur Seite, die 128 Bit groß sind, die Synergistic Processing Elements (SPE). Abbildung 4.3[10] zeigt den schematischen Aufbau des Cell-Prozessors. Der Element Interconnect Bus (EIB), der die Kerne verbindet, arbeitet in Form von vier gegenläufigen Ringen, was in Abbildung 4.4 verdeutlicht werden soll. Mit ihm sind auch der Speicher-Controller (Mem-C.) und andere Schnittstellen (I/O) verbunden.

Zwischen dem PPE und den SPEs können Signale oder Nachrichten, die in Mailboxen abgelegt werden, ausgetauscht werden, Datentransfers per DMA sind ebenfalls zwischen allen Kernen möglich [39]. Die Kerne des Cell-Prozessors greifen auf streng getrennte Bereiche des Arbeitsspeichers zu. Da die Trennung lediglich über

[6] Zur Unterscheidung von „Power Architecture“ und „PowerPC“ siehe http://en.wikipedia.org/w/index.php?title=Power_Architecture&oldid=385277212

[7] http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=P4080

[8] Zum Beispiel von Curtiss-Wright: http://www.cwcembedded.com/products/0/3/676.html.

[9] 1500,- bis 4000,- $ (Stand Oktober 2010), siehe http://www.freescale.com/files/32bit/doc/brochure/PWRARCHQIQSG.pdf

[10] modifiziert nach http://de.wikipedia.org/w/index.php?title=Datei:Schema_Cell.png&filetimestamp=20061009180747

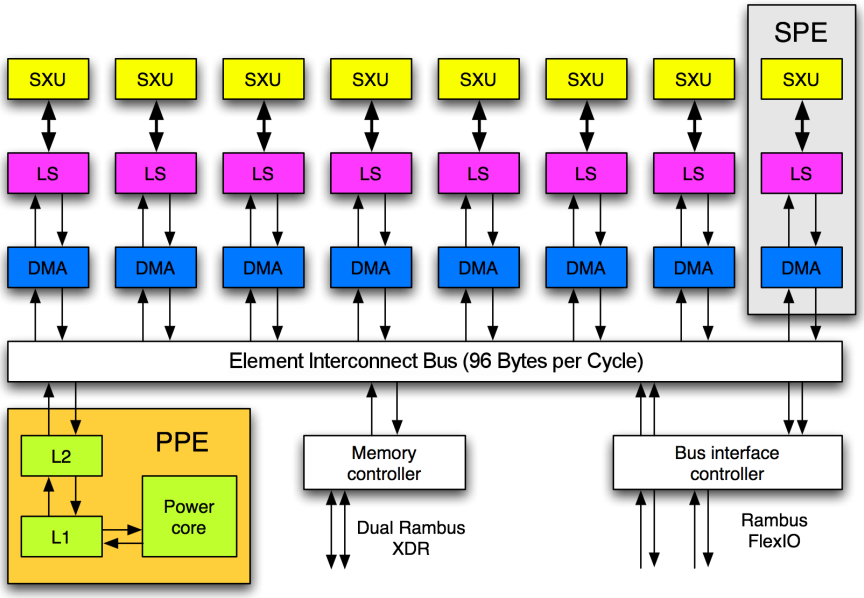


Abbildung 4.3.: Strukturschema der Cell-CPU

die Adressierung erfolgt, sind bei DMA-Transfers also Behinderungen denkbar, falls diese nicht aufeinander abgestimmt werden. Allerdings verfügt jedes SPE über 256 KByte lokalen Speicher für Daten und Instruktionen, größere Transfers können also als Sonderfall betrachtet beziehungsweise vermieden werden, um möglichst vorhersagbares Verhalten zu erreichen.

Der Cell-Prozessor ist kommerziell verfügbar, unter anderem wird er in einer Spielkonsole[11] und in Steckkarten für PCs[12] verbaut.

4.4.3. Cavium OCTEON

Die OCTEON-Familie der Firma Cavium beherbergt Prozessoren mit 4 bis zu 32 Kernen in MIPS-Architektur. MIPS-Systeme sind in eingebetteten Systemen häufig anzutreffen [64]. Die OCTEON-Prozessoren sind auf Kommunikationssysteme wie Netzwerk-Router ausgelegt und daher mit zahlreichen Schnittstellen ausgestattet. Informationen über Scratchpad-RAMs oder Kommunikationsmechanismen sind bislang nicht öffentlich zugänglich[13].

[11] Sony PlayStation 3, erhältlich zwischen 300,- und 600 €. (Stand September 2010)
[12] mvXCell - 8i von Matrix Vision ca. 3500,- €. (Stand September 2010) Produktseite: `http://www.matrix-vision.com/products/cell/mvxcell8i.php?lang=de`
[13] `http://www.caviumnetworks.com/OCTEON_MIPS64.html`

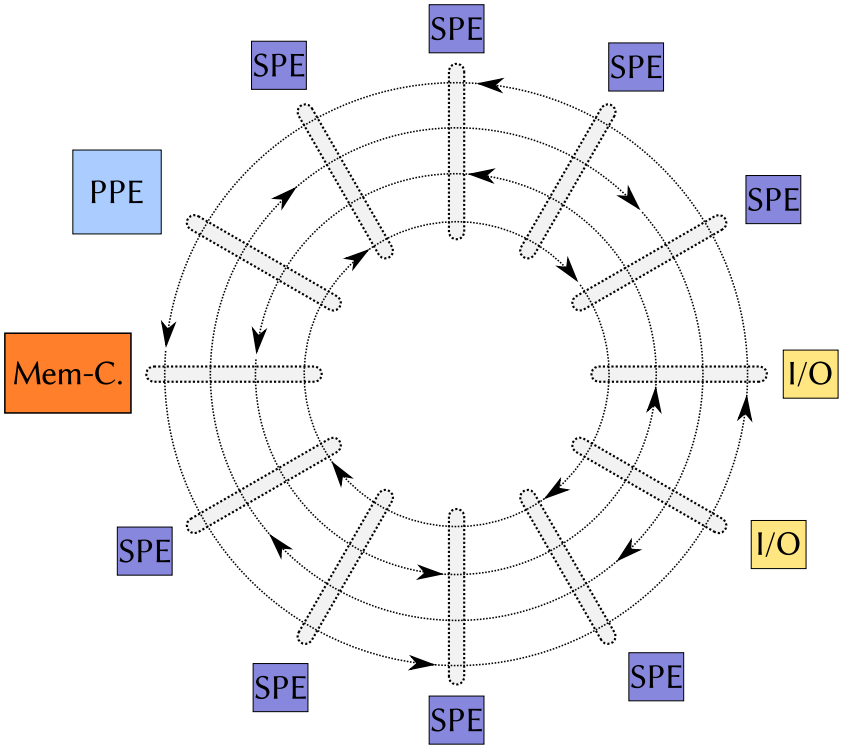


Abbildung 4.4.: Kommunikationsschema der Cell-CPU

4.4.4. ARM Cortex-A9 MPCore und ARM11 MPCore

Prozessoren mit ARM-Architektur werden von Lizenznehmern der ARM Holdings produziert und sind daher unter verschiedensten Markennamen erhältlich. Sie sind die häufigsten in eingebetteten Systemen: Jährlich werden mehrere Milliarden verkauft[14], insbesondere in Mobiltelefonen kommen mittlerweile ganz überwiegend energieeffiziente ARM-CPUs zum Einsatz[15]. Die zahlreichen Modelle basieren auf unterschiedlichen Versionen der ARM-Architektur, was die Orientierung erschwert.

Die Prozessoren der ARM-Cortex-A-Reihe wie der Cortex-A9 MPCore[16] sind multicore-fähige Kerne in ARMv7-Architektur und werden von ARM als „application processors“ bezeichnet. Die ARM11-MPCore-Prozessoren gehören hingegen zur weniger leistungsfähigen[17] ARM11-Architektur, die aber explizit für den Embedded-Bereich vorgesehen ist. Mit Kernen beider Architekturen sind Prozessoren mit bis zu vier Kernen möglich.

[14] siehe *ARM Holdings plc Reports Results for the Second Quarter and Half Year Ended 30 June 2010* (27. Juli 2010)
[15] http://techon.nikkeibp.co.jp/NEA/archive/200204/177680/
[16] http://www.arm.com/products/processors/cortex-a/cortex-a9.php
[17] http://www.arm.com/products/processors/index.php

Zur Gewährleistung der Cache-Kohärenz wurden die Prozessoren mit einer „Snoop Control Unit“ ausgestattet[18], über den Interrupt Controller kann ein Kern einem anderen mitteilen[19], dass eine Nachricht im Shared Memory abgelegt wurde [30]. In Abbildung 4.5 ist schematisch die Anordnung von Caches, „Snoop Control Unit“ und Interrupt-Controller dargestellt. Bei beiden Architekturen können Caches einzeln abgeschaltet werden.

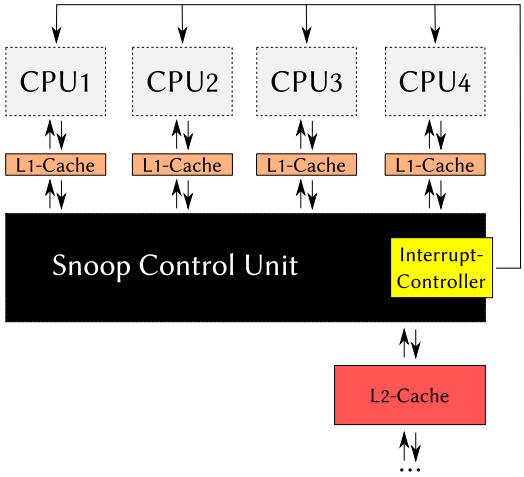


Abbildung 4.5.: Kommunikation innerhalb ARM-CPUs, schematisch

Der NaviEngine1 von NEC ist ein auf Navigationssysteme ausgerichteter Prozessor mit vier ARM11-Kernen und eingebauter Graphics Processing Unit (GPU), der bis zu 1920 MIPS leistet[20]. Das einzige dem Autor bekannte Board mit dieser CPU kostet mehr als ein voll ausgestatteter Desktop-PC[21]. Boards mit Cortex-A9-Prozessoren und mehr als zwei Kernen waren zum Zeitpunkt dieser Arbeit nicht erhältlich.

4.5. IP-Cores

Einige Prozessorkerne liegen nur als IP-Core vor. Als IP-Core bezeichnet man elektronische Schaltungen, die nur in einer Hardwarebeschreibungssprache wie VHDL vorliegen und auf einem FPGA implementiert werden. Ein Field Programmable Gate Array (FPGA) ist ein Chip, der in Blöcken organisierte Logikgatter und andere Schaltungen enthält, deren Verbindung und Verhalten programmiert werden kann. Der Begriff „IP-Core“ kommt von „Intellectual Property“ – der Kern ist das „geistige Eigentum“ des Herstellers. Bei der Erstellung eines Designs können mehrere IP-Cores

[18] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0360f/I1000175.html
[19] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0048a/Beibdbaa.html
[20] http://www.arm.com/markets/embedded_solutions/armpp/24826.html
[21] Ca. 2700,- € (Stand Oktober 2010), siehe http://www.t-engine4u.com/archive/mp_t-kernel_ne1.pdf (japanisch)

zusammengefügt und der Prozessor so um zusätzliche Komponenten oder eigene Schaltkreise erweitert werden. Aus der entstandenen VHDL-Beschreibung wird eine Netzliste generiert, ein Map-and-Route-Algorithmus passt anschließend die Netzliste für den konkreten FPGA an. Dabei fallen Informationen über das mögliche Zeitverhalten und den Platzbedarf an. Ist das Ergebnis akzeptabel, wird eine Binärdatei zum Beschreiben des PROMs des FPGA generiert. Die notwendige Software für den Map-and-Route-Schritt und zur Erzeugung der Binärdatei muss in der Regel vom Hersteller des FPGA käuflich erworben werden, proprietäre IP-Cores werden oft nur als fertige Netzliste ausgeliefert, so dass sie nicht leicht eingesehen oder verändert werden können.

Gibt es von Seiten des Herstellers kein entsprechendes Design, so kann es notwendig sein, ein Multicoresystem aus mehreren IP-Cores selbst zu entwerfen. Da dies sehr aufwändig werden kann, werden nur solche IP-Cores berücksichtigt, die explizit mit Multicore-Fähigkeiten daherkommen.

4.5.1. Xilinx Microblaze- und PowerPC-Kerne

Beim Microblaze handelt es sich um einen proprietären IP-Core des FPGA-Herstellers Xilinx. Mit dem „Base System Builder“ der Software „ISE Design Suite“ von Xilinx kann ein Zweikernsystem erstellt werden[22], weitere Kerne sind möglich, wenn der Platz auf dem FPGA ausreicht. Der RISC-Kern verfügt über Mechanismen zum gegenseitigen Ausschluss (Mutual Exclusion (MUTEX)), Mailboxen und Fast Simplex Link (FSL), einen unidirektionalen Punkt-zu-Punkt-Kommunikationskanal. OpenFire, eine freie Alternative, die nur einen Teil des Befehlssatzes des Microblaze beherrscht, wurde am Virginia Polytechnic Institute entwickelt [19], besitzt aber kein Speicherinterface.

FPGAs vom Typ Virtex-5 FXT können maximal zwei PowerPC-Kerne enthalten[23]. Ein asymmetrisches Design aus PowerPC- und Microblaze-Kernen würde den Rahmen dieser Arbeit sprengen.

4.5.2. Aeroflex Gaisler LEON3

Der LEON3 basiert auf einem ursprünglich für die Europäische Weltraumorganisation ESA entwickelten Design von 32-Bit-Prozessoren in SPARC-V8-Architektur [26, 1]. Er steht als Teil der GRLIB IP Library[24] unter GNU General Public License (GPL) zur Verfügung. Mit der GRLIB kann ein vollständiges SoC erzeugt werden, sie enthält Module für Schnittstellen wie CAN, SPI, I²C, Ethernet, USB, PCI und darüber hinaus verschiedene Speichercontroller und Scratchpad-RAM. Der LEON3

[22] MicroBlaze v7.20 FAQ: http://www.xilinx.com/products/design_resources/proc_central/microblaze_faq.pdf

[23] Virtex-5 Family Overview: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf – Virtex-5 FXT FPGAs: http://www.xilinx.com/products/virtex5/fxt.htm

[24] http://www.gaisler.com/cms/index.php?option=com_content&task=section&id=13&Itemid=125

unterstützt SMP-Betrieb mit bis zu 16 Kernen, einzelne Komponenten wie Interruptcontroller, Fließkommaeinheit oder Caches können abgeschaltet werden, um Platz auf dem FPGA zu sparen [24]. Message-Passing gibt es aber nicht.

Mit dem LEON3-FT existiert eine Version, die fehlertolerant und daher besonders für den Raum- und Luftfahrtbereich geeignet ist [25]. Sie steht aber, ebenso wie der Nachfolger LEON4, nicht unter einer freien Lizenz. Boards mit FPGAs, für die in der GRLIB bereits Designs existieren und die groß genug sind, mehrere LEON3-Kerne aufzunehmen, sind verhältnismäßig teuer[25]. Ein Emulator existiert, ist aber nicht frei verfügbar.

Programme für den LEON3 können in C geschrieben und mit BCC, einer freien Erweiterung des GCC, kompiliert werden. Mit dem kostenlosen Debug-Monitor GRMON[26], der eine Verbindung zum GNU Debugger (GDB) herstellen kann, ist Lock-Step-Debugging möglich.

4.6. MPSoCs

Die folgenden fertigen MPSoCs benötigen nahezu keine externe Hardware, sie sind lauffähige Ein-Chip-Systeme.

4.6.1. Freescale MSC8156

Der Freescale MSC8156 ist eigentlich ein DSP mit 6 Kernen, von denen jeder über einen 512kB großen, zu einem Scratchpad-RAM umschaltbaren L2-Cache verfügt. Laut Dokumentation[27] kann er aber mit einem Echtzeit-Betriebssystem betrieben werden, er ist also universell programmierbar.

4.6.2. Parallax Propeller

Der Parallax Propeller [46] ist ein mit 8 RISC-Kernen, sogenannten *Cogs*, ausgestattetes MPSoC und leistet bis zu 160 MIPS. Jeder Kern verfügt über 2kB eigenen Arbeitsspeicher. Außerdem steht ein gemeinsam genutzter Speicher von 32kB zur Verfügung. Der Zugriff auf diesen wird durch einen Hub umlaufend jeweils nur einem Kern zur Zeit gestattet, was Konkurrenzsituationen und damit indeterministisches Verhalten verhindert. Interrupts gibt es nicht. Der Propeller kann heruntergetaktet werden, aber alle Kerne laufen mit dem selben Takt.

In Abbildung 4.6 ist die Propeller-Architektur gezeigt, die schematische Darstellung des Hubs unten rechts im Bild.

[25] GR-XC3S mit Xilinx Spartan3 XC3S1500: 750,- €,
GR-PCI-XC5V mit Xilinx Virtex 5 XC5VLX50: 3450,- € (Stand September 2010).

[26] http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=39&Itemid=128

[27] http://cache.freescale.com/files/dsp/doc/ref_manual/MSC8156RM.pdf?fpsp=1

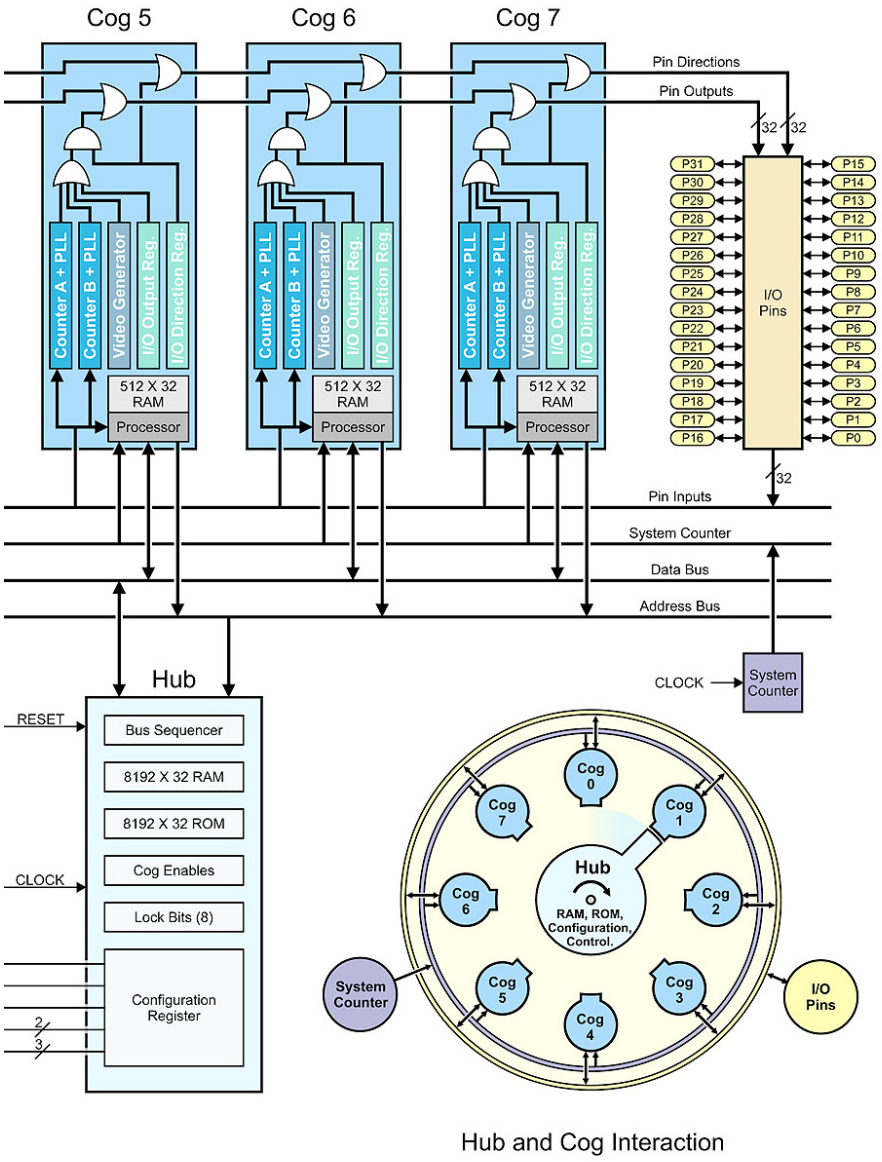


Abbildung 4.6.: Die Propeller-Architektur (Ausschnitt aus: Propeller Manual [46])

Der Propeller ist im Hobby-Bereich etabliert, wohl auch aufgrund des geringen Preises[28]. Es existiert eine rege Entwicklergemeinschaft[29], der Einstieg in die Programmierung kann also als einfach angesehen werden. Typisches Einsatzgebiet ist der Modellbaubereich; so existiert mit dem „octopilot“[30] zum Beispiel ein mit einem Propeller umgesetzter Autopilot.

[28] Evaluation Boards kosten unter 100,-€. (Stand September 2010)
[29] http://forums.parallax.com/forumdisplay.php?f=65
[30] http://code.google.com/p/octopilot/

Der Prozessor wird in der eigens entwickelten Hochsprache „Spin“ programmiert, es existieren aber auch ein kommerzieller[31] und ein freier[32] C-Compiler.

4.6.3. XMOS XS1-G4

Prozessoren des Herstellers XMOS enthalten bis zu vier RISC-Kerne der hauseigenen XS1-Architektur [47]. Die Kerne sind SMT-fähig, sie führen bis zu acht Hardware-Threads aus, welche normalerweise umlaufend (*round-robin*) geschedulet werden. Threads können aber auch auf Events oder Interrupts warten. Wartende Threads verbrauchen keine Rechenleistung.

Die XS1-Prozessoren enthalten Hardware-Synchronizer und Timer sowie ein internes, mittels eines Switches partitionierbares Netzwerk, über das Nachrichten zwischen den Kernen ausgetauscht werden können. Da Schnittstellen des Netzwerks nach außen geführt sind, können weitere Prozessoren angebunden werden, der Nachrichtenaustausch ist dann direkt zwischen allen Kernen des Systems möglich. Laut Spezifikation [48] garantiert die Partitionierung Echtzeitverhalten.

Jeder Kern verfügt über je 64kB SRAM, einen gemeinsamen Speicher gibt es bis auf einen kleinen PROM und eventuell per SPI angebundenen Flash-RAM nicht. Weitere Speicher müssen per Software-Controller angebunden werden und sind daher sehr langsam[33].

Abbildung 4.7[34] zeigt den Aufbau des XS1-G4. Der XMOS XS1-G4 erreicht mit 32 Threads maximal 1600 MIPS. Unter den verfügbaren Kits sind relativ preisgünstige mit Ethernet-Schnittstellen[35]. Eine Eclipse-basierte Entwicklungsumgebung ist kostenlos verfügbar, außerdem erzeugt der freie Compiler Low Level Virtual Machine (LLVM) Maschinencode für XMOS-Prozessoren, die in einer C-Erweiterung namens XC programmiert werden. Der Hersteller verspricht mit XC deterministisches Echtzeitverhalten[36].

4.7. Mesh-Prozessoren

Mesh-Prozessoren enthalten eine Vielzahl von im Gitter angeordneten Kernen. Je nach Ausführung können die Kerne nur mit ihren direkten Nachbarn kommunizieren oder über ein internes Netzwerk mit allen. Mesh-Prozessoren sind extrem leistungsfähig, setzen aber eine Programmierung voraus, die diese spezielle Struktur ausnutzt.

[31] http://elmicro.com/de/iccprop.html
[32] http://catalina-c.sourceforge.net/
[33] Zum Beispiel der SDRAM-Treiber: http://www.xmoslinkers.org/projects/sdram
[34] Entnommen aus: XS1-G4 512BGA Datasheet:
http://www.xmos.com/published/xs1-g4-512bga-datasheet-0
[35] 80,- bis 150,- € inklusive Zubehör. (Stand September 2010)
[36] http://www.xmos.com/technology/xc

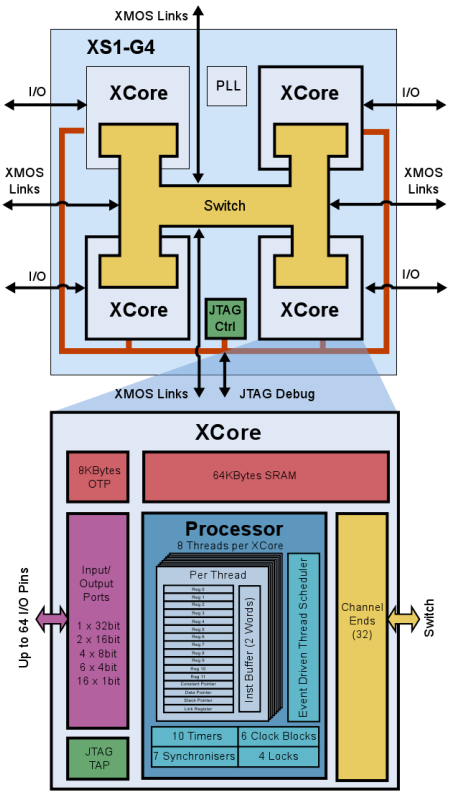


Abbildung 4.7.: XMOS XS1-G4 Vierkernprozessor

4.7.1. Ambric Am2045

Der Ambric Am2045 wird als Massively Parallel Processor Array (MPPA) bezeichnet und besteht aus 336 RISC-Kernen, die über FIFO-Channels kommunizieren[37]. Er wird in Java programmiert, ein Map-Route-Algorithmus verteilt voneinander abgeschottete Rechen-Objekte automatisch auf dem System. Der Ambric ist außergewöhnlich energieeffizient [57], aber MPPAs sind keine typischen eingebetteten Systeme. Das System ist außerdem de facto ein GALS und somit nicht hinreichend vorhersagbar.

4.7.2. Tilera TILE64

Der TILE64 hat 64 in einem Gitter angeordnete, voll ausgestattete Kerne, die über eigene Caches mit zwei Ebenen, aber keinen eigenen Speicher verfügen[38]. Ein internes Netzwerk ermöglicht die Kommunikation der Kerne untereinander, es ist auf hohen Datendurchsatz ausgelegt; Tilera-Prozessoren sind auf Netzwerkanwendungen und Video-Streaming ausgerichtet. Bemerkenswert ist die Cache-Architektur: Inhal-

[37] http://www.ambric.com/technologies_mppa.php

[38] http://www.tilera.com/sites/default/files/productbriefs/PB010_TILE64_Processor_A_v4.pdf

te können von benachbarten Kernen gelesen werden[39]. Ein zyklengenauer Simulator existiert, ist aber ebensowenig wie die Entwicklungsumgebung frei verfügbar. In Abbildung 4.8[40] ist der prinzipielle Aufbau des Gitters zu sehen.

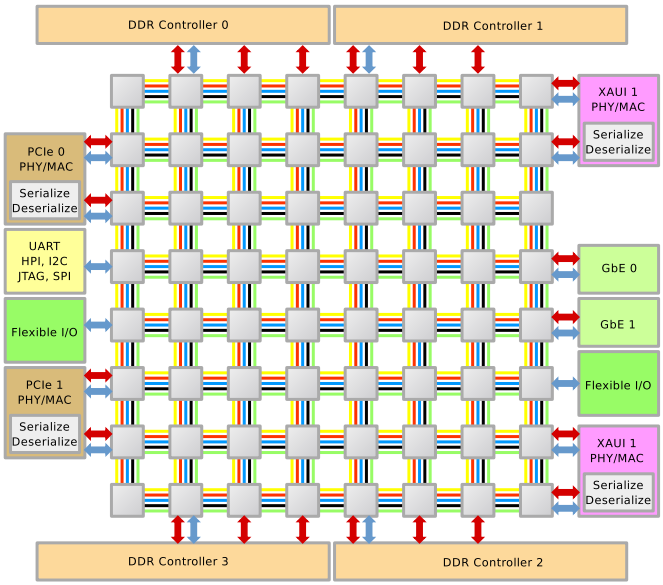


Abbildung 4.8.: Struktur des Tilera TILE64

4.7.3. IntellaSys SEAforth 40C18

Der SeaForth-Prozessor[41] ist wie der TILE64 ein Mesh-Prozessor. Allerdings können die 40 Kerne des 40C18 nur mit ihrem jeweiligen Nachbarn kommunizieren, und das, da ein gemeinsamer Takt fehlt, nur asynchron[42]. Die Schnittstellen des Prozessors, den Adress- und Datenbus eingeschlossen, können nur von einigen Kernen genutzt werden[43]. Die Speicher der einzelnen Kerne sind winzig, ROM und RAM speichern je nur 64 Worte. Verarbeitet werden nur Instruktionen in einer eigenen Sprache namens VentureForth. Entwicklungsumgebungen sind für Linux und Windows frei verfügbar, ein Simulator existiert. IntellaSys gibt als Einsatzbereiche unter anderem batteriebetriebene Anwendungen an, was aufgrund des geringen Stromverbrauchs im Milliamperebereich glaubwürdig erscheint.

[39] http://www.linuxfordevices.com/c/a/News/Massively-multicore-processor-runs-Linux/
[40] Bild von Wikipedia-User „MovGP0“ unter CC-BY-SA 3.0 http://en.wikipedia.org/wiki/File:Tile64.svg
[41] http://www.intellasys.net/index.php?option=com_content&task=view&id=60&Itemid=75
[42] http://www.intellasys.net/index.php?option=com_content&task=view&id=21&Itemid=41
[43] http://www.intellasys.net/templates/trial/content/S40C18_PB080408.pdf

4.8. Wissenschaftlich-experimentelle Plattformen

Einige Prozessoren werden nicht kommerziell entwickelt und hergestellt, sondern von Forschungsgruppen, die besondere Anforderungen zu erfüllen versuchen.

4.8.1. EMPEROR

Der EMPEROR ist ein Multiprozessor-Projekt der University of Auckland [20, 66]. Er basiert auf modifizierten PIC-CPUs[44], genannt Reactive PIC (RePIC), deren Instruktionssatz so erweitert wurde, dass sie Teile der Esterel-Semantik direkt unterstützen [16]. Von diesen können bis zu 32 über eine *Thread-Control-Unit (TCU)* gekoppelt werden, die eine Barrier für die Tick-Synchronisierung zur Verfügung stellt und den Austausch von Signalen ermöglicht. Die TCU enthält auch Funktionen zum Threadhandling und unterstützt Kommandos wie `FORK` oder `JOIN`.

Die EMPEROR-Architektur ist in Abbildung 4.9 dargestellt.

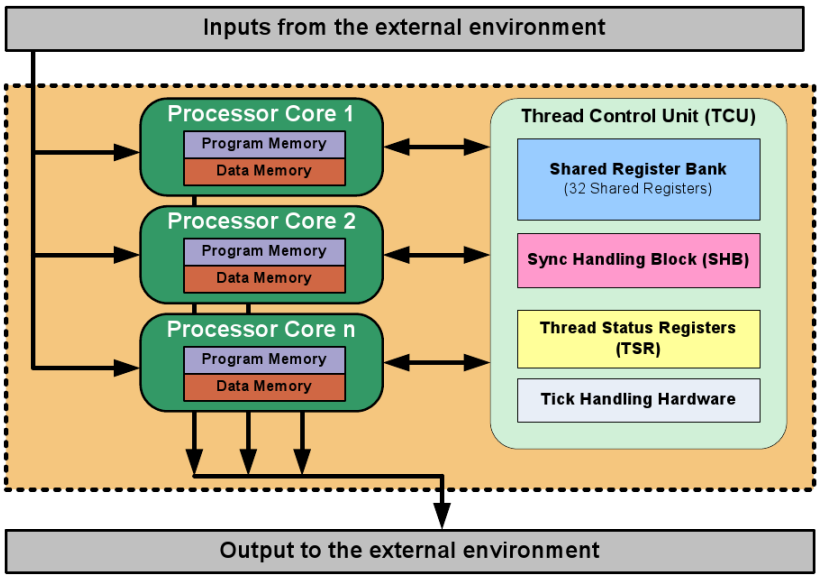


Abbildung 4.9.: Die EMPEROR-Architektur (aus: Dayaratne et al. [20])

Der EMPEROR muss auf einem FPGA implementiert werden, ein Simulator existiert nicht. Er ist mehr ein Multiprozessor-System als Multicore-Prozessor, und wie bereits in Kapitel 2 erwähnt, existiert für den EMPEROR lediglich ein Esterel-Compiler.

[44] http://en.wikipedia.org/w/index.php?title=PIC_microcontroller&oldid=386813140

4.8.2. Berkeley PRET Architecture

Die in Abschnitt 4.2 genannten Probleme haben Edwards et al. [23] motiviert, eine neue Prozessorarchitektur mit wiederholbarem Timing vorzuschlagen, die Precision Timed Architecture (PRET) und einen auf der SPARC-V8-Architektur [1] basierenden Prozessor zu entwerfen [43], welcher in dieser Arbeit einfach PRET genannt wird.

Der PRET ist ein SMT-Prozessor und arbeitet seine Threads in einer Art Zeitmultiplexverfahren ab. Alle Threads bekommen gleich viel Rechenzeit zugewiesen und werden unabhängig von der ausgeführten Instruktion round-robin geschedulet. Anstatt zu blockieren, zum Beispiel weil ein Thread auf Daten aus dem Hauptspeicher wartet, wird die entsprechende Instruktion des Threads immer wieder ausgeführt, bis sie erfolgreich war. Durch diesen „Replay-Mechanismus“ werden die anderen Threads weiter ausgeführt und Blockaden vermieden.

In der Standardkonfiguration ist der Hauptspeicher in 8 MB Shared Memory und 1 MB privaten Speicher für jeden Thread aufgeteilt. Der Speicher kann vergrößert werden. Jedem Thread steht außerdem statt eines Caches ein 64 kB großer Scratchpad-RAM mit einer Zugriffszeit von einem Taktzyklus für Instruktionen und Daten zur Verfügung. Die Scratchpad-RAMs können durch softwaregesteuerte DMA-Transfers Daten aus und in den Hauptspeicher übertragen. Sonstige Speicherzugriffe erfolgen über ein sogenanntes *Memory-Wheel*, einen Mechanismus, der mit dem Hub des Propeller-Chips vergleichbar ist. Das vereinfachte Blockdiagramm des PRET in Abbildung 4.10 zeigt den Aufbau des Speichers.

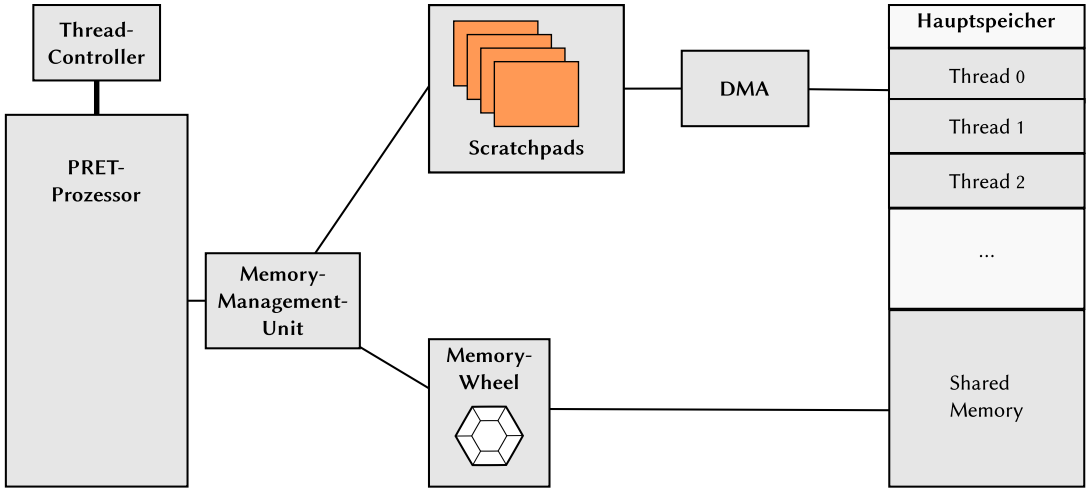


Abbildung 4.10.: Blockdiagramm der PRET-CPU, vereinfacht (nach Lickly et al. [43])

Das Memory-Wheel garantiert atomaren und fairen Speicherzugriff, um aber deterministisches Verhalten zu erreichen, muss die Reihenfolge von Zugriffen festgelegt werden können. Zu diesem Zweck wurde der Befehlssatz um eine *Deadline*-Instruktion erweitert [36]. Die Deadline-Instruktion bestimmt den frühesten Zeitpunkt, zu dem ein Befehl ausgeführt werden soll, bezogen auf letzte Rücksetzung

des Zählers. Mit der Deadline-Instruktion lässt sich also die Periode einer Schleife festlegen. Startet man Schleifen in verschiedenen Threads mit unterschiedlicher Verzögerung und sorgt dann für eine identische Periode, so kann man sicherstellen, dass Speicherzugriffe in einer bestimmten Reihenfolge stattfinden [43].

Der PRET ist somit mit einem SMP-Prozessor mit zusätzlichen Synchronisationsmechanismen vergleichbar. Sein Verhalten ist perfekt vorhersagbar, und er ist schneller, als ein Prozessor ohne Multithreading-Unterstützung mit sechs Threads wäre, aber natürlich langsamer als ein Prozessor, der mehrere Instruktionen gleichzeitig ausführen kann.

Bislang besitzt der PRET keine Funktionalität zur direkten Kommunikation der Threads untereinander, allerdings garantiert das Memory-Wheel gegenseitigen Ausschluss, so dass, geschicktes Timing vorausgesetzt, Shared-Memory-Kommunikation zumindest ohne Race Conditions möglich ist.

Der PRET kann auf einem FPGA implementiert werden, es existiert aber auch ein zyklengenauer Simulator[45].

4.9. Weitere Multicore-Systeme

Einige Multicore-Prozessoren wurden bereits während der Recherchen zu diesem Abschnitt als ungeeignet erkannt und daher nicht näher betrachtet. Der Vollständigkeit halber sollen sie kurz genannt werden:

- Der von Lickly et al. [43] erwähnte RAW-Prozessor des MIT befindet sich noch im Prototypenstadium.
- Die in zahlreichen Varianten erhältlichen Prozessoren von Texas Instruments, wie der TMS320C80 MVP und der Analog Devices Blackfin BF561 sind Kombinationen aus voll ausgestattetem Prozessorkern und DSP.
- Der Infineon Danube hat lediglich zwei Kerne. Hingegen haben der ASOCS ModemX 128, die picoChip-Prozessoren 200 bis 300 und der IBM Kilocore 1025 Kerne.
- Die Azul Systems Vega-Prozessoren führen nur Java aus.
- Ferner bietet Broadcom Kommunikationsprozessoren mit vier Kernen an, die restriktive Informationspolitik des Unternehmens macht diese jedoch unattraktiv.

4.10. Zusammenfassung und Fazit

In den Tabellen 4.1 und 4.2 sind, sofern bekannt, die wichtigsten Eigenschaften der in den Abschnitten 4.4 bis 4.8 genannten Prozessoren aufgeführt. Die Vielfalt auf

[45] http://chess.eecs.berkeley.edu/pret/src/pret-1.0/pret_simulator.html

Prozessor	Form	Kerne/Threads	SMT	SMP	Scratchpad o.ä.
Parallax Propeller	MPSoC	8		x	2kB
Ambric AM2045	Karte	336			x
ARM Cortex-A9 MPCore	Chip	2 - 4		x	
ARM11 MPCore	Chip	2 - 4		x	
Berkeley PRET	Simulator/FPGA	6	x		64kB
Cavium Octeon	Chip	4 - 32		x	
Freescale QorIQ 4080	Chip	8			
Freescale MSC8156 (DSP)	MPSoC	6		x	x
Gaisler LEON3	IP/FPGA	4 - 16		x	x
IBM/Sony/Toshiba Cell	Chip	1 + 8		x	x
Xilinx Microblaze	IP/FPGA	2 - n		x	
Xilinx PowerPC IP auf Virtex-5	IP/FPGA	2		x	
XMOS XS1-G4	MPSoC	4 (x8)	x	x	64kB
Tilera TILE64	Chip	64		x	
IntellaSys SEAforth 40C18	Chip	40		x	64 Worte

Tabelle 4.1.: Prozessoreigenschaften, Teil 1

Prozessor	Architektur	Anordnung	Kommunikation	Compiler/IDE	Sprachen
Parallax Propeller	RISC	-	Shared Memory	Spin kostenlos, C proprietär und & frei	Spin, (C)
Ambric AM2045	RISC	MPPA	FIFO-Kanäle	Eclipse-basiert	Java, Asm.
ARM Cortex-A9 MPCore	ARM RISC	-	Shared Memory	GCC	C
ARM11 MPCore	ARM RISC	-	Shared Memory	GCC	C
Berkeley PRET	SPARC/PRET	-	Shared Memory	GCC	C
Cavium Octeon	MIPS	-			
Freescale QorIQ 4080	Power	-	Message-Passing	proprietär	C
Freescale MSC8156 (DSP)	„StarCore“	-			
Gaisler LEON3	SPARC V8	-	Shared Memory	BCC (GCC), Eclipse-basiert	C
IBM/Sony/Toshiba Cell	Power + SPE	Ring/Stern	DMA, Mailboxen	GCC	C
Xilinx Microblaze	RISC	-	Mutexe, FSL, Mailboxen	Eclipse-basiert	C
Xilinx PowerPC IP auf Virtex-5	Power	-			C
XMOS XS1-G4	RISC	-	Message-Passing, Synchronizer	Eclipse-basiert + LLVM	C / XC
Tilera TILE64	?	Mesh			C
IntellaSys SEAforth 40C18	RISC (Forth)	Mesh	zwischen benachbarten	proprietäre IDE	VentureForth

Tabelle 4.2.: Prozessoreigenschaften, Teil 2

dem Prozessormarkt ist enorm, das gilt erst recht für Prozessoren für eingebettete und Echtzeitsysteme. Deren Eigenschaften überschneiden sich, so dass eine Einordnung oft nicht eindeutig möglich ist. Das Beispiel des Freescale MSC8156 zeigt, dass selbst ein als DSP beworbener und mit speziellen Recheneinheiten ausgestatteter Chip gleichzeitig ein MPSoC mit für ein Betriebssystem ausreichendem Befehlssatz sein kann. Unterschiede im Detailreichtum der Beschreibungen sind ebenfalls festzustellen. Belastbare Informationen über Eigenschaften und Leistungsfähigkeit sind selten; die Hersteller haben indes kein Interesse daran, die negativen Eigenschaften ihrer Produkte in den Vordergrund zu stellen.

Die Einordnung in Abbildung 4.11 ist daher letztendlich ein wenig willkürlich und richtet sich nach der aus Sicht des Autors am meisten hervorstechenden Eigenschaft der jeweiligen CPU, wie zum Beispiel der Anordnung der Kerne (Ringbus, Arrays und Meshes). Die ovalen roten (bzw. dunkelgrauen) Felder stellen diese Kategorien dar, die eckigen roten (bzw. dunkelgrauen) benennen die Architektur (ARM, MIPS). Die eckigen gelben (bzw. hellgrauen) Felder enthalten die Namen der Hersteller. Im oberen Zweig stellt der Propeller den einzigen kommerziell verfügbaren Prozessor dar, der EMPEROR stammt aus einem wissenschaftlichen Projekt, bietet aber kein wiederholbares Timing.

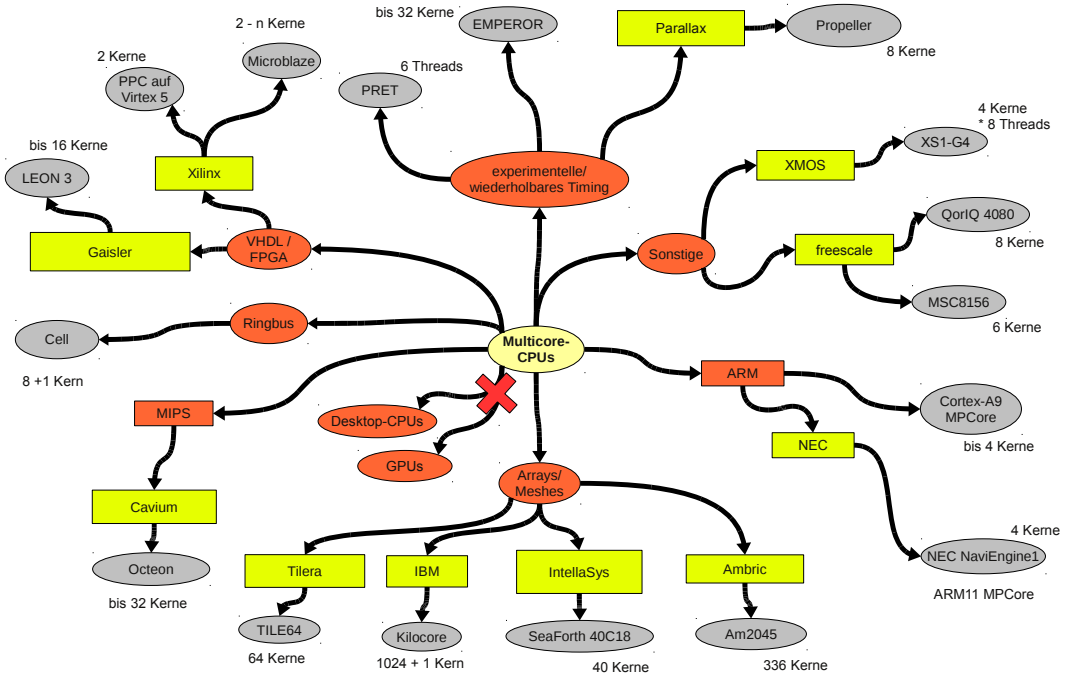


Abbildung 4.11.: Einteilung der CPUs als Baum

Mesh-Prozessoren und MPPAs wie die von Ambric, Tilera und IntellaSys im unteren Teil von Abbildung 4.11 haben für diese Arbeit zu viele Kerne. Während die Prozessoren von Ambric und Tilera schlichtweg überdimensioniert sind, sind

die Kerne des IntellaSys zu klein und können nicht in C programmiert werden. Asynchrone Kommunikation der Kerne untereinander ist bei der Umsetzung eines synchronen Systems zudem ebensowenig förderlich wie die Cache-Architektur des Tilera TILE64. Der als IP-Core erhältliche Microblaze ist ein proprietärer Kern, für wissenschaftliche Arbeiten werden ihm andere daher vorgezogen, die Alternative OpenFire ist ohne Speicher-Interface eher für Datenflussanwendungen geeignet. Der Parallax Propeller bietet einen leichten Einstieg in die Implementierung reaktiver Systeme, der sehr kleine Speicher schränkt aber zu sehr ein. Der XMOS XS1-G4 ist deutlich leistungsfähiger und hat zahlreiche Eigenschaften, die ihn für die synchrone Programmierung interessant machen, insbesondere Hardware-Synchronizer und Message-Passing über den prozessorinternen Switch. Beiden fehlt aber ebenfalls ein Speicher-Interface.

Plattformen mit Freescales QorIQ 4080 sind teuer und der Prozessor gehört eher zu den auf Kosten der Vorhersagbarkeit hochgezüchteten. Bei ihm stört darüber hinaus ebenso wie beim MSC8156 die proprietäre Entwicklungsumgebung.

Die Cell ist als Steckkarte ebenfalls sehr teuer. Als Alternative böte sich eine PlayStation 3 mit Linux an, wie sie Edwards und Vasudevan [58] verwendet haben. Nach der Entwicklung des SHIM-Backends sind sie jedoch zu der Ansicht gelangt, dass für heterogene Multicore-Plattformen höhere Sprachen als C genutzt werden müssen [21], was für diese Arbeit zunächst ein starkes Argument gegen solche komplexen Plattformen darstellt. Seit Firmware-Version 3.21 ist außerdem kein Linux-Betrieb der PlayStation 3 mehr möglich[46].

ARM bietet zwar bereits mehrere Architekturen mit Multicore-Fähigkeiten an, diese haben aber keine Mechanismen zur direkten Kommunikation der Kerne untereinander. ARM hat neben der Cache-Kohärenz zwar auch die Kommunikation zwischen den Prozessorkernen als Flaschenhals erkannt [30], aber Nachrichten werden im gemeinsam genutzten Speicher abgelegt und durch einen Interrupt signalisiert, was die empfangende CPU in ihrer Ausführung unterbricht. In Hinblick auf die synchrone Programmierung eines Systems ist das ein Hindernis. Für Experimente geeignete Plattformen mit Multicore-Prozessoren von ARM sind zur Zeit nicht zu bekommen.

Der LEON3 ist als quelloffene, leicht zugängliche und umfangreich konfigurierbare SPARC-CPU mit Lock-Step-Debugging-Fähigkeit eine gut geeignete Plattform. Die fehlende Message-Passing-Funktionalität stört auch hier, eine spätere Nachrüstung ist aber denkbar, da der LEON3 auf einem FPGA implementiert wird. Versuche mit verschiedenen FPGAs verliefen jedoch nicht zufriedenstellend. Bereits ein Dualcore-System ist für einen Xilinx Virtex-4-FX12 zu groß. Ein vorhandenes Board mit dem größeren Xilinx Spartan-3-1500 ließ sich nach dem Programmieren des PROMs nicht mit dem Debugger GRMON kontaktieren. Die Details inklusive Statistiken zur Auslastung der FPGAs sind in Anhang D aufgeführt.

[46] Artikel *It no longer does everything: no more Linux on PlayStation 3* bei ars technica, 29. März 2010 `http://tinyurl.com/yd3yr2o`

Der EMPEROR basiert auf der wenig leistungsfähigen PIC-Architektur, ist auf Esterel ausgelegt und vor allem nicht frei verfügbar.

Der ebenfalls aus wissenschaftlichen Arbeiten hervorgegangene Precision Timed Architecture ist in C universell programmierbar. Zwar bietet auch er kein Message-Passing. Dank des frei verfügbaren Simulators ist allerdings ein sofortiger Einstieg in die Programmierung möglich. Durch sein Zeitmultiplex-Scheduling, den Scratchpad-RAM und die resultierende Tatsache, dass sich Threads nicht gegenseitig beeinflussen, kommt er einem SMP-Prozessor ausreichend nahe, um als Plattform für diese Arbeit zu dienen. Die Kooperation des hiesigen Lehrstuhls mit den anderen am PRET-Projekt beteiligten Arbeitsgruppen und die neuartige Speicherarchitektur sprechen ebenfalls für ihn. Daher werden einige Implementierungen von SCmc auf dem PRET-Simulator realisiert.

5. SC auf Multicore

Synchronous C (SC) wurde für diese Arbeit um Kommunikations- und Synchronisationsmechanismen erweitert, die die Parallelausführung mehrerer SC-Programme ermöglichen. In diesem Kapitel werden in Abschnitt 5.2 und 5.3 die zugrundeliegenden Konzepte erläutert, die SC zu Synchronous C auf Multicore (SCmc) erweitern. Im Anschluss wird in Abschnitt 5.4 auf Hierarchieabhängigkeiten gesondert eingegangen, bevor die beiden Methoden verglichen werden. Außerdem werden in Abschnitt 5.6 Ansätze zur Erzeugung von parallelem Code aus SyncCharts genannt, die zur Modifikation des SC-Compilers genutzt werden könnten. Zuletzt wird ein Formalismus betrachtet, mit dem sich zeigen lässt, dass die Parallelausführung bestimmter Aktionen erlaubt ist.

Begrifflichkeiten

Da diese Bezeichnung bei der Beschreibung von verteilten Systemen üblich ist, wird auch hier von *Locations* gesprochen. Damit sind Orte gemeint, an denen Code ausgeführt wird und die über getrennte Speicherbereiche verfügen. Eine Location kann ein Prozessorkern sein, ein Systemprozess, der einen Kern simuliert, oder auch ein Hardware-Thread. Eine Location kann also mehrere Software-Threads ausführen. Sind letztere gemeint, wird einfach nur von Threads gesprochen. Ein Thread, der ein Signal sendet, wird als Sender oder *Emitter* bezeichnet. *Lokal* ist, was auf derselben Location liegt oder stattfindet. Da PIDs in SC Prioritäten entsprechen, werden beide Begriffe unterschiedslos verwendet.

5.1. Einleitung

Bisherige Arbeiten zur Verteilung synchroner Programme betrachten überwiegend verteilte Systeme. Das heißt, die Locations sind vergleichsweise weit entfernt, agieren autonom [14, S. 523ff] und sind über asynchrone Kommunikationskanäle verbunden. Nur selten ist dabei eine Leistungssteigerung das Ziel. Bei Datenflusssprachen wird zum Teil eine Desynchronisierung versucht, die es erlaubt, Long-Duration-Tasks (LDTs) aus den schnelleren Teilen des Programms auszukoppeln. Eine Ausdehnung des synchronen Bereichs wird hingegen nicht versucht.

Die von Girault [27] genannten Arbeiten suchen Nebenläufigkeit in sequentiellem Zwischencode, also nachdem die dem ursprünglichen Programm inhärente Nebenläufigkeit entfernt wurde. Die verteilten Programme müssen im Nachhinein resynchronisiert werden.

In der Regel wird davon ausgegangen, dass die Parallelität in einer synchronen Sprache der einfacheren und modulareren Beschreibung des Systems dient und von der beabsichtigten Implementierung abweichen kann [15].

Mit dieser Arbeit wird hingegen die Möglichkeit zur Parallelausführung von SC-Programmen untersucht. Damit direkt verbunden ist die Nutzung der inhärenten Parallelität eines SyncCharts. Auch wenn Nebenläufigkeit im Design nicht immer mit Laufzeit-Parallelität gleichzusetzen ist, kann man vermuten, dass ein Programmierer in einigen Fällen durchaus eine parallele Ausführung beabsichtigt. Das könnte insbesondere dann der Fall sein, wenn ein Programm aus mehreren Modulen zusammengesetzt wurde, und wenn eine grafische Modellierungssprache die Parallelität direkt sichtbar macht. Solange Abhängigkeiten korrekt berücksichtigt werden, spricht zumindest nichts gegen diese Vorgehensweise.

Die hier vorgestellten Synchronisationsmethoden stützen sich auf die Tatsache, dass ein Multicore-System im Vergleich zu einem verteilten System eine enge Kopplung über sehr schnelle und zuverlässige Kanäle oder einen Shared Memory erlaubt.

Eine grundlegende Erkenntnis muss aber auch bei der Verteilung auf Prozessorkerne beachtet werden: Da die Kausalitätsanalyse modulares Kompilieren eines synchronen Programms verbietet [27], muss das zu verteilende Programm als Ganzes betrachtet werden. Es darf nur *während* des Kompilierens in interagierende Teile zerlegt werden. Voraussetzung für deren korrekte Funktion und dafür, dass ihr Verhalten dem des zentralen Programms entspricht, ist, dass der Compiler Synchronisationsmechanismen einbaut oder Annahmen über vorhandene Mechanismen machen kann. Daher sollen diese Mechanismen, soweit praktikabel, so in SCmc eingebaut werden, dass ein Compiler ein Programm möglichst einfach zerlegen kann. Dies macht Compilermodifikationen deutlich einfacher.

Darüber hinaus sollen sowohl die SC-Syntax als auch der eigentliche SCmc-Code möglichst beibehalten oder nur geringfügig erweitert werden. Prinzipiell ließe sich zwar eine minimal erweiterte Signalkommunikation nutzen, um direkt in SC einen Synchronizer zu implementieren. Das würde aber den Programmcode aufblähen und schlecht lesbar machen. Die vorgestellten Methoden sollen daher in die SCmc-Makros eingebaut werden und für den Programmierer transparent sein. Eine Ausnahme stellen hier die Operatoren `FORK` und `JOIN` dar, die im SCmc-Code durch spezielle Konstrukte ergänzt werden.

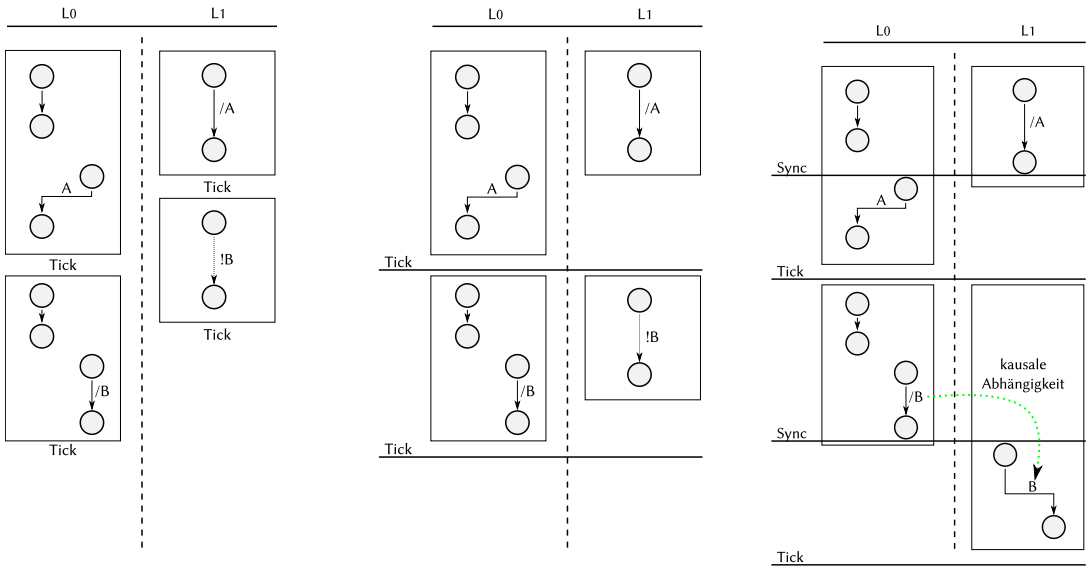
Ansätze zur Synchronisation

Der in Abschnitt 3.4 beschriebene Compiler setzt kausale Abhängigkeiten im SyncChart in Prioritäten um. Der Dispatcher entscheidet anhand dieser über die Ausführungsreihenfolge, so dass die Abhängigkeiten zur Laufzeit immer korrekt berücksichtigt werden.

Abbildung 5.1 zeigt gleichzeitige Ausführungen zweier Teile eines synchronen Programms auf verschiedenen Locations. Die Kästen umrahmen die Abläufe während jeweils eines Ticks. Die Zustände der Threads sind als Kreise dargestellt, der Übergang von einem in einen anderen Zustand desselben Threads als Pfeil. Abgesehen

von einem Broadcast der Signale über alle Locations sei zunächst keine Wechselwirkung angenommen. Die Locations haben außerdem getrennte Speicherbereiche.

Einfach auf allen Locations die lokalen Threads mit den jeweils höchsten Prioritäten auszuführen, ist offenbar keine Lösung. In Abbildung 5.1(a) führt die völlige Unabhängigkeit der Locations voneinander zu einem Fehler (rot gepunktet). Location 1 hat den zweiten Tick beendet, bevor auf Location 0 das Signal `B` gesendet werden konnte. Es ist naheliegend, die Ticks zu synchronisieren, wie in Abbildung 5.1(b) gezeigt, doch kann der Fehler auch dann auftreten. Ein sequentiell ausgeführtes Programm hätte sichergestellt, dass die Überprüfung des Status von `B` erst nach dessen möglicher, im Beispiel tatsächlicher, Aussendung stattfindet, wie in 5.1(c) gezeigt.



(a) Unabhängige Ausführung (b) Ticksynchronisation (c) Feingranulare Synchronisation

Abbildung 5.1.: parallele Ausführungen

Zur feingranulareren Synchronisation wie in Abbildung 5.1(c) wurden im Rahmen dieser Arbeit zwei unterschiedliche Ansätze verfolgt:

Signalbasierte Synchronisation: Code, der kein Signal prüft, ist auch nicht von anderem Code abhängig, kann also parallel ausgeführt werden. Wird ein Signal geprüft, so wird gewartet, bis dessen Zustand sicher bekannt ist.

Prioritätenbasierte Synchronisation: Code mit gleicher Priorität darf gleichzeitig ausgeführt werden. Die Prioritäten werden global kommuniziert.

Dabei spielen die folgenden grundlegenden Makros eine besondere Rolle:

PRIO: Ein Thread kann seine Priorität wechseln. Es muss sichergestellt sein, dass die neue Priorität belegt werden kann und dass der Wechsel die Synchronisation nicht negativ beeinflusst.

FORK: Werden Threads zur nebenläufigen Ausführung abgespalten, so darf auch diese Veränderung nicht zu Fehlern führen.

JOIN: Die Synchronisation darf nicht dazu führen, dass ein JOIN vor den abgespaltenen Child-Prozessen ausgeführt wird.

ABORT: Ebenso müssen Weak Abortions und Strong Abortions korrekt behandelt werden. Ein abgebrochener Prozess darf außerdem keine Blockaden verursachen.

Damit ein Signal überall im System wahrgenommen wird, muss außerdem mindestens das Makro EMIT angepasst werden, für die signalbasierte Synchronisation auch PRESENT. Die genaue Realisierung des Signal-Broadcasts hängt aber von der Art der Synchronisation ab.

In den nächsten beiden Abschnitten werden die beiden Ansätze zur Synchronisation erläutert und dabei die Auswirkungen auf die genannten Makros betrachtet. Dabei werden Wechselwirkungen von FORK, JOIN und ABORT über mehrere Locations hinweg zunächst ausgeklammert.

5.2. Signalbasierte Synchronisation

Werden Threads auf verschiedenen Locations gleichzeitig ausgeführt, so kann ein Thread nicht mehr ohne weiteres feststellen, dass ein Signal ABSENT ist. Die Signale werden daher als dreiwertig behandelt: Zu PRESENT und ABSENT kommt noch der Status „unbekannt“, dargestellt als $\perp$. Prioritäten haben bei der signalbasierten Synchronisation nur noch lokale Bedeutung. Es können also auf verschiedenen Locations gleiche Prioritäten belegt sein oder Threads mit unterschiedlichen Prioritäten gleichzeitig ausgeführt werden.

5.2.1. Die Certainty-Matrix

Ein Signal gilt als ABSENT, wenn bewiesen werden kann, dass es im laufenden Tick nicht mehr gesendet werden wird. Findet die Ausführung verschiedener Threads nicht in einer festgelegten Reihenfolge statt, so geht die dazu notwendige Information verloren. Zähler vorzuhalten, die die Zahl der potentiellen Emitter im laufenden Tick enthalten, ist nicht sinnvoll. Der konkurrierende Zugriff auf einen solchen Zähler müsste mit Konstrukten zum gegenseitigen Ausschluss geregelt werden. Dadurch könnten sich Threads unnötig gegenseitig beeinflussen.

Konkurrierender Zugriff kann vermieden werden, indem die Information über potentielle Emitter in einer Matrix im Shared Memory verwaltet wird. In dieser wird für jeden Thread und jedes Signal gespeichert, ob bekannt ist, dass der Thread das Signal im laufenden Tick nicht mehr senden wird. Die Matrix ergänzt die Datenstruktur, in der die Zustände PRESENT und ABSENT gespeichert werden. Steht ein Signal dort auf ABSENT, so wird es zunächst als $\perp$ angenommen, wenn nicht für alle Threads ein ausschließender Eintrag in der Matrix vorhanden ist. Ist letzteres der Fall, so ist es CERTAINLY ABSENT [5], daher wird die Matrix *Certainty-Matrix* genannt.

```
1  TICKSTART(4);
2
3  ABO:
4     FORK(AB, 1);
5     FORKE(ABOMain);
6
7  AB:
8     FORK(WaitA, 2);
9     FORK(WaitB, 3);
10    FORKE(ABMain);
11
12 WaitA:
13    AWAIT(A);
14    TERM;
15
16 WaitB:
17    AWAIT(B);
18    TERM;
19
20 ABMain:
21    JOIN;
22    EMIT(O);
23    TERM;
24
25 ABOMain:
26    AWAIT(R);
27    TRANS(ABO);
28
29 TICKEND;
```

(a) ABRO

PID	A	B	R	O
1	1	1	1	0
2	1	1	1	1
3	1	1	1	1
4	1	1	1	1

(b) Certainty-Matrix für ABRO

Abbildung 5.2.: ABRO und die zugehörige Certainty-Matrix

Da ein Thread zur Laufzeit schon vor Ende seiner Ausführung in der Certainty-Matrix eintragen kann, welche Signale er nicht mehr senden wird, steht diese Information anderen Threads früher zur Verfügung. Sie müssen also nicht immer warten, bis alle potentiellen Emitter eines Signals ihre Ausführung für den laufenden Tick abgeschlossen haben. Ist im Vorfeld bekannt, welche Threads welche Signale niemals oder zumindest nicht in bestimmten Ticks aussenden, so kann die Matrix vor jedem Tick mit entsprechenden Einträgen initialisiert werden. Durch die Certainty-Matrix können also voneinander abhängige Threads in Teilen parallel ausgeführt werden, was einen Geschwindigkeitsgewinn erwarten lässt.

Der Certainty-Matrix kommt die Architektur des PRET sehr zugute. Jeder Hardware-Thread auf einem PRET kommt innerhalb von 77 Zyklen einmal zu einem Speicherzugriff [43], egal ob lesend oder schreibend. Das Memory-Wheel garantiert gegenseitigen Ausschluss und atomaren Speicherzugriff.

Darüber hinaus greift jeweils nur ein Thread schreibend auf ein Feld der Certainty-Matrix zu, und es gibt nur zwei mögliche Einträge: `1` (sicher) und `0` (unsicher). Grundsätzlich darf ein Eintrag während des laufenden Ticks nur von `0` auf `1` geändert werden. Da wiederholt gelesen wird, spielt die Reihenfolge von Lese- und Schreibzugriffen keine Rolle. Wurde ein von `0` auf `1` geänderter Eintrag kurz zuvor noch als `0` gelesen, so wird dies beim nächsten Lesevorgang bemerkt. Race-Conditions sind also ausgeschlossen, und es tritt kein Readers-Writers-Problem [18] auf.

Zum Beispielprogramm ABRO in Listing 5.2(a) ist in Abbildung 5.2(b) die Certainty-Matrix mit den a priori bekannten Einträgen für die Signale `A`, `B`, `R` und `O` gezeigt. Bei ABRO kann nur das Signal `O` überhaupt gesendet werden, und dies ist nur im Zustand `ABMain` möglich, also in Thread 1. Daher enthält die Matrix im Feld in Zeile 1 und Spalte `O` eine `0`. Alle anderen Einträge können bereits mit `1` initialisiert werden.

Damit die Certainty-Matrix zur Synchronisation zwischen verschiedenen Locations benutzt werden kann, muss für jeden möglichen Thread auf jeder Location eine Zeile existieren. Der Einfachheit halber wird auf jeder Location das gleiche Maximum an Threads Max_{PID} angenommen, so dass ($\text{Max}_{\text{PID}} \cdot \text{Locations}$) Zeilen angelegt werden müssen.

Wird ABRO, wie in den Listings 5.3(a) und 5.3(b) gezeigt, auf zwei Locations verteilt, so entsteht die Certainty-Matrix in Abbildung 5.3(c). Da es vier Signale gibt und auf zwei Locations maximal drei Threads laufen, werden 24 Felder benötigt. Dass nur je drei PIDs benötigt werden, wurde im Beispiel bereits berücksichtigt.

Damit die Felder in der Matrix auch dann nicht von `1` auf `0` geändert werden, wenn ein Thread mit `PRIO` die Priorität und damit die PID wechselt, werden diese beim Schreiben indirekt adressiert. Beim Prioritätswechsel werden nur die Adressen der alten und neuen PID in einem lokalen Array vertauscht, die Matrix selbst ändert sich nicht. Die indirekte Adressierung wird in Abbildungen der Übersichtlichkeit halber weggelassen.

Stellt nun ein Thread beim Prüfen eines Signals fest, dass es nicht PRESENT ist und sich in der zugehörigen Spalte in der Certainty-Matrix irgendwo eine `0` befindet, so muss er warten. Der Status des Signals ist $\bot$. Sein eigenes Feld ist davon natürlich ausgenommen, das kann spätestens bei der Prüfung zu `1` gesetzt werden.

Während der Thread wartet, könnte er selbst von der Ausführung zurückgestellt werden und die Ausführung anderer Threads zulassen oder die gesamte Location blockieren.

```
1  TICKSTART(3);
2
3  ABO:
4     FORK(AB, 1);
5     FORKE(ABOMain);
6
7  AB:
8     FORK(WaitA, 2);
9     FORKE(ABMain);
10
11 WaitA:
12    AWAIT(A);
13    TERM;
14
15 ABMain:
16    JOIN;
17    EMIT(O);
18    TERM;
19
20 ABOMain:
21    AWAIT(R);
22    TRANS(ABO);
23
24 TICKEND;
```

(a) Location 0

```
1  TICKSTART(3);
2
3  ABO:
4     FORK(AB, 1);
5     FORKE(ABOMain);
6
7  AB:
8     FORK(WaitB, 2);
9     FORKE(ABMain);
10
11 WaitB:
12    AWAIT(B);
13    TERM;
14
15 ABMain:
16    JOIN;
17    EMIT(O);
18    TERM;
19
20 ABOMain:
21    AWAIT(R);
22    TRANS(ABO);
23
24 TICKEND;
```

(b) Location 1

PID		A	B	R	O
L0	1	1	1	1	0
	2	1	1	1	1
	3	1	1	1	1
L1	1	1	1	1	0
	2	1	1	1	1
	3	1	1	1	1

(c) zugehörige Certainty-Matrix

Abbildung 5.3.: Verteiltes ABRO und Certainty-Matrix

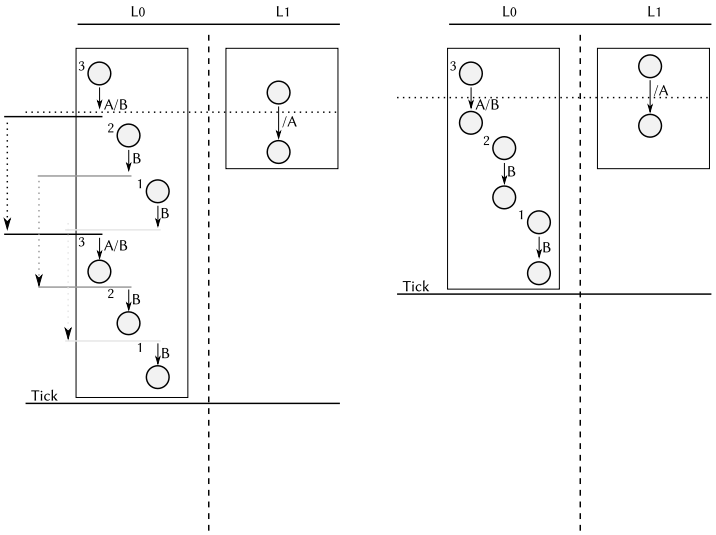
5.2.2. Blockierende Threads

Zum Blockieren von einzelnen Threads werden ein zusätzlicher Thread-Zustand sowie die dazugehörige Datenstruktur benötigt. Dies macht den Prioritätswechsel mit `PRIO` noch ein wenig teurer, als er im Verhältnis zu anderen Operatoren schon ist. Der Dispatcher muss den zusätzlichen Zustand bei der Auswahl des nächsten auszuführenden Threads berücksichtigen. Er kann blockierte Threads zum Beispiel erst dann erneut ausführen, wenn es keine aktiven Threads mehr gibt oder aber gleich nachdem ein einzelner aktiver Thread abgearbeitet wurde. In jedem Fall müssen blockierte Threads wie beim in Kapitel 2 beschriebenen EMPEROR so lange ausgeführt werden, bis sie die Blockade überwinden und ihre Ausführung für den Tick beenden können.

Durch dieses dynamische Scheduling wirken Prioritäten bestenfalls noch optimierend, haben aber keinen zwingenden Einfluss mehr auf die Abläufe. Die genaue Ausführungsreihenfolge hängt zudem stark vom zeitlichen Verhalten des Systems, mithin von Zufällen, ab. In Abbildung 5.4(a) blockiert beispielsweise ein Thread mit der Priorität 3, weil das Signal `A` auf einer anderen Location zu spät gesendet wird (horizontale gepunktete Linie). Andere Threads mit niedrigeren Prioritäten sind von ihm über das Signal `B` abhängig, daher werden diese ihren Prioritäten nach bis zur Signalprüfung ausgeführt und, weil auch `B` den Status $\perp$ hat, sukzessive blockiert.

Die dickeren, grau abgestuften Linien und die weit gepunkteten Pfeile markieren die Zeiträume, über die die Threads in ihrer Ausführung unterbrochen werden.

Wenn der ursprüngliche Thread mit der Priorität 3 wieder an der Zeile ist, wird die Blockade durch das Signal `B` gelöst. Bis alle Threads abgearbeitet sind und der Tick beendet werden kann, kann es deutlich länger dauern, als wenn das Signal, wie in Abbildung 5.4(b) gezeigt, schon früher PRESENT gewesen wäre. Es fällt zusätzlicher Scheduling-Overhead an, dazu kommt der Aufwand der erfolglosen Signalprüfung. Allerdings ist auch denkbar, dass der Aufwand gerechtfertigt ist, weil während der Zeit bis zum Eintreffen des Signals `A` durch die teilweise Abarbeitung anderer Threads Zeit gewonnen wird. Das hängt vom konkreten System und dem Rechenaufwand in den Zuständen der Threads ab.



(a) ungünstiger Verlauf (b) günstiger Verlauf

Abbildung 5.4.: Ausführungen mit blockierenden Threads

Davon unabhängig sind die Auswirkungen auf die in der Einleitung genannten Operatoren fatal: Wenn ein Thread seine Priorität mit `PRIO` wechseln möchte, dann muss die neue Priorität frei sein. Ist sie belegt, weil der sie belegende Thread blockiert, so muss der erste ebenfalls blockieren, bis die Priorität frei ist. Alternativ könnte man das Makro `PRIO` einfach wirkungslos machen.

Neben Signalabhängigkeiten und blockierten Prioritäten müssen aber auch noch Abbruchbedingungen berücksichtigt werden. Während beim EMPEROR Abbruchbedingungen von jedem Thread geprüft werden, werden sie in SC in einen einzelnen Zustand ausgelagert, dessen Wirkung vom Scheduling abhängt. Setzt dieser Thread eine Strong Abortion um, so muss er vor seinen Child-Threads ausgeführt werden. Blockiert er, müssen also alle seine Child-Threads ebenfalls blockieren. Ebenso muss sichergestellt werden, dass `JOIN`s und mögliche Weak Abortions nach den Child-Threads ausgeführt werden. Es muss also in `JOIN` und `ABORT` geprüft werden, ob

nicht noch Child-Threads unter den blockierten Threads sind und ob sie eine höhere oder niedrigere Priorität haben.

Um all dies zu berücksichtigen, sind tiefgreifende und komplizierte Erweiterungen notwendig, die letztendlich nur abbilden, was in den Prioritäten in SC schon an Vorwissen steckt. Die Flexibilität der Ausführung wird mit indeterministischem Scheduling und stärker schwankenden Ausführungszeiten erkauft. Dies lässt den Ansatz wenig erfolgversprechend erscheinen.

5.2.3. Blockierende Locations

In Abbildung 5.5 ist eine Ausführung gezeigt, die bis zum Eintreffen des Signals `A` vollständig unterbrochen wird. Hier wird, wenn ein geprüftes Signal den Status $\perp$ hat, der Dispatcher nicht angesprungen. Stattdessen behält der wartende Thread die Kontrolle und prüft das Signal, bis dessen Status sicher bekannt ist. Alle nachfolgenden Threads werden von der Ausführung abgehalten, die gesamte Location blockiert also.

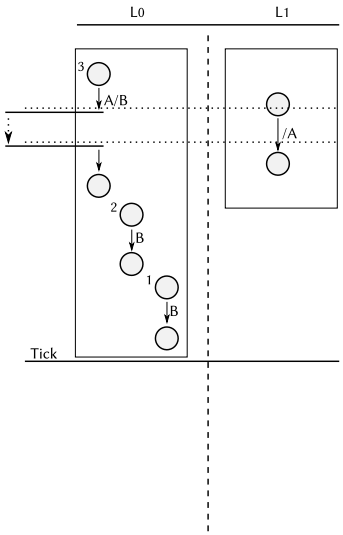


Abbildung 5.5.: Ausführung mit blockierender Location

Bei dieser Methode können die Prioritäten ihre Wirkung zumindest lokal weiterhin entfalten. Zur Compile-Zeit bekanntes Wissen wird mit einer Ausnahme von der Regel, ein Thread dürfe nur in seine eigene Zeile in der Certainty-Matrix schreiben, genutzt: Blockiert die Location wegen eines Signals `S`, so kann kein anderer Thread derselben Location `S` senden. Der müsste als potentieller Emitter schließlich eine höhere Priorität haben und bereits ausgeführt worden sein. `S` kann also von der eigenen Location aus nicht mehr gesendet werden, der blockierende Thread darf daher „senkrecht“ in die Matrix schreiben und `S` lokal als sicher markieren.

```
1  FORKER1:
2    forked=0;
3    if (PRESENT(X)) {
4      FORK(ABEMITTER, 7);
5      forked=1;
6    }
7    else if (PRESENT(Y)) {
8      FORK(CDEMITTER, 7);
9      forked=1;
10   }
11   if (forked) FORKE(SOMESTATE);
12   EMIT(Z);
13   PRIO(8);
14   ...
15
16 ABEMITTER:
17   ...
18   if (PRESENT(C)) EMIT(A);
19   if (...) EMIT(B);
20   ...
21
22 CDEMITTER:
23   ...
24   if (...) EMIT(C);
25   if (...) EMIT(D);
26   ...
```

(a) FORK unterschiedlicher Emitter

PID	A	B	C	D	X	Y	Z
1	1	1	1	1	1	1	0
⋮							
7	?	?	?	?	1	1	1
8	1	1	1	1	1	1	1

(b) Certainty-Matrix

Abbildung 5.6.: Unvorhersehbare Initialisierung

Da lokale Abhängigkeiten automatisch berücksichtigt werden, gibt es keine direkten Probleme mit JOIN, ABORT oder PRIO. Weil aber im Gegensatz zum im vorigen Abschnitt beschriebenen Verfahren nicht jeder Thread garantiert irgendwann ausgeführt wird, muss die Matrix korrekt initialisiert werden.

Kann die Certainty-Matrix nicht im Vorfeld für jeden Tick mit korrekten Werten initialisiert werden, sind komplizierte Verfahren notwendig. Das ist beispielsweise dann der Fall, wenn es von einem Eingabesignal abhängt, ob ein Thread mit FORK erzeugt wird oder nicht. Listing 5.6(a) zeigt einen Ausschnitt aus einem Programm, das die PID 7 unterschiedlich oder gar nicht belegt, während die Belegung für PIDs 1 und 8 klar ist. Einige Einträge in der Certainty-Matrix in Abbildung 5.6(b) sind daher unvorhersehbar, dargestellt durch Fragezeichen.

Nullinitialisierung und Deadlock

Wie in 5.2.1 erklärt, dürfen Einträge während des laufenden Ticks nur von `0` auf `1` geändert werden. Es scheint also naheliegend, die Certainty-Matrix in Teilen oder bei a priori unbekannter Anzahl von Threads und Prioritätenbelegungen vollständig mit `0` zu initialisieren. Während des Ticks können die Threads selbst ihre Einträge machen. Unbenutzte Zeilen können komplett mit `1` gefüllt werden.

Die Nullinitialisierung kann jedoch zu einem Deadlock führen, wie in Abbildung 5.7 gezeigt. Listing 5.7(a) zeigt einen Ausschnitt aus einem Programm, welches se-

```
1  T4:
2    if (PRESENT(X)) ...
3
4  T3:
5     EMIT(Y);
6     ...
7
8  T2:
9     if (PRESENT(Y)) ...
10
11 T1:
12    calculate();
13    TERM;
```

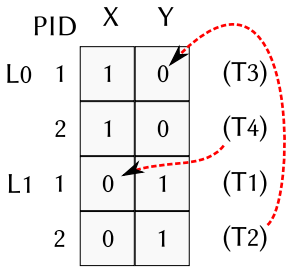
(a) sequentielles Programm

```
1  T4:
2    if (PRESENT(X)) ...
3
4  T3:
5     EMIT(Y);
6     ...
```

(b) Location 0

```
1  T2:
2     if (PRESENT(Y)) ...
3
4  T1:
5     calculate();
6     TERM;
```

(c) Location 1



(d) Certainty-Matrix

Abbildung 5.7.: Deadlock durch Nullinitialisierung der Certainty-Matrix

quentiell ausgeführt zu keinerlei Problemen führt. Zur leichteren Nachvollziehbarkeit sind die Threads nach ihren Prioritäten benannt und absteigend geordnet.

Wird das Programm verteilt, so dass Location 0 `T4` und `T3` ausführt und Location 1 `T2` und `T1`, so entsteht eine Certainty-Matrix wie in Abbildung 5.7(d). Rechts neben dieser sind die Namen der Threads, links die neuen lokalen Prioritäten angegeben.

Die beiden Threads mit der jeweils höchsten Priorität werden ausgeführt, im Beispiel also `T4` und `T2`. Location 0 blockiert, weil `T4` darauf wartet, dass für Signal `X` eingetragen wird, dass Thread `T1` es nicht senden wird. Durch die Initialisierung der Matrix mit `0` steht das Wissen, dass das niemals geschehen wird, zur Laufzeit nicht zur Verfügung.

Von seiner eigenen Location kann `T4` wegen der in den lokalen Prioritäten immer noch partiell abgebildeten kausalen Abhängigkeiten wissen, dass `X` nicht in später ausgeführten Threads gesendet wird, die entsprechenden Einträge in der Matrix also auf `1` setzen. Über die andere Location sind jedoch keine Annahmen möglich, schließlich könnte dort genauso gut ein Thread mit höherer Priorität, der vielleicht noch ein `X` sendet, laufen.

Location 1 blockiert wiederum, weil `T4` die Ausführung von `T3` verhindert, so dass nicht entschieden ist, ob Signal `Y` gesendet wird. Da der dort aktive Thread `T2` lediglich `Y` prüft, werden die Einträge für `X` nicht verändert, `T4` blockiert also weiterhin beziehungsweise beide Locations sich gegenseitig.

Die Abhängigkeiten sind in Abbildung 5.7(d) und 5.8 durch rote, gepunktete Pfeile verdeutlicht. Die nullinitialisierte Certainty-Matrix hat offensichtlich in SCmc eine zirkuläre Abhängigkeit eingeführt, die in SC unmöglich war. Es kommt zu einem Deadlock. Selbst wenn `T2` und `T4` zu Beginn des Ticks mitgeteilt hätten, dass sie `Y` und `X` nicht senden werden, bliebe die Abhängigkeit bestehen.

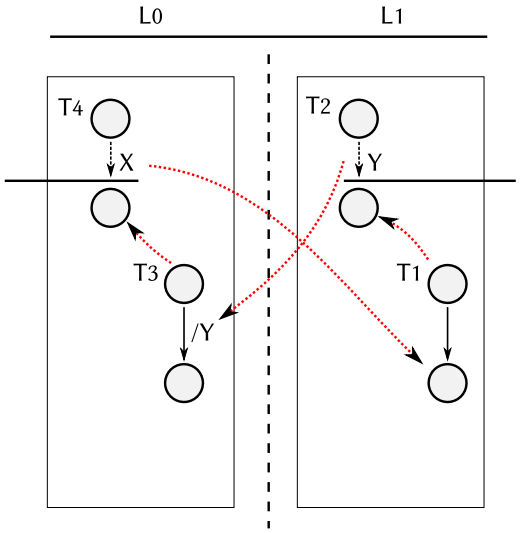


Abbildung 5.8.: Zirkuläre Abhängigkeit

Hinzu kommt, dass es in der Certainty-Matrix wegen ihrer festen Größe immer unbenutzte Zeilen geben kann. Wenn im Vorfeld nicht bekannt ist, welche Zeilen unbenutzt sind und während des kommenden Ticks auch *bleiben*, so müssen diese während des Ticks „ausgeeinst“ werden, damit andere Locations nicht daran blockieren. Der einzige Ort, an dem das geschehen kann, ist das Makro `FORKE`, das einen oder mehrere `FORK`s abschließt. Das allerdings dürfte *höchstens* einmal pro Tick geschehen. Mehrere Zustände mit `FORK`s und `FORKE`s dürften nicht im selben Tick ausgeführt werden, weil bereits im ersten `FORKE` bislang unbenutzte Zeilen mit `1` gefüllt worden wären. Ein SyncChart wie in Abbildung 5.9 wäre – zumindest für weniger als vier Locations – unübersetzbar.

Umgekehrt müsste aber in jedem Tick auch *mindestens* einmal „ausgeeinst“ werden, auch wenn keine neuen Threads abgespalten wurden, da die nullinitialisierten unbenutzten Zeilen sonst Blockaden verursachen würden. Wenn keine `FORK`s und somit auch kein `FORKE` ausgeführt werden, kommt es jedoch niemals dazu. Ein einfaches Beispiel ist ein System, in dem alle Prozesse gerade in einem `AWAIT` stecken. Abgesehen von den Signalen, auf die sie warten, können die Prozesse auf ihren eigenen Locations nichts als sicher markieren. Informationen darüber, ob andere Threads im laufenden Tick `FORK`s ausführen werden, haben sie ebenfalls nicht.

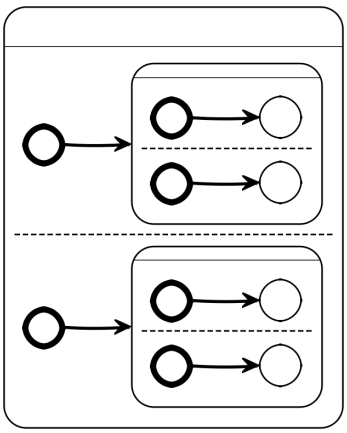


Abbildung 5.9.: SyncChart mit mehrfacher Aufspaltung im selben Tick

Deadlocks erkennen mit dem LCR-Algorithmus

Prinzipiell kann man Deadlocks erkennen und sogar auflösen. Dazu eignet sich eine leichte Abwandlung des asynchronen LCR-Algorithmus [45, S. 477] zur Anführerwahl in einem Ring, der hier in einer abgewandelten Form vorgestellt werden soll. Zunächst sei angenommen, dass bei der Verteilung des Programms auf mehrere Locations eindeutige PIDs beibehalten wurden.

Definition (LCR-Algorithmus (informell, abgewandelt))**.** *Jede blockierende Location sendet die PID des lokalen blockierenden Threads p_l an die Location ✗, von der sie abhängig ist, das heißt, die die Blockade aus ihrer Sicht verursacht. Wenn eine Location blockiert und eine PID p_r empfängt, verfährt sie wie folgt:*

$p_l < p_r$*: Ist die empfangene PID p_r größer als p_l, so wird p_r an ✗ gesendet.*

$p_l > p_r$*: Ist die empfangene PID p_r kleiner als p_l, so wird p_r verworfen und p_l (erneut) an ✗ gesendet.*

$p_l = p_r$*: Ist die empfangene PID mit der lokalen identisch, so wird angenommen, dass das Signal, dessen Status benötigt wird, von ✗ nicht mehr emittiert wird.*

Von welcher anderen eine Location abhängig ist, kann leicht am Index der Zeile in der Certainty-Matrix, in der das geprüfte Signal als $\bot$ erkannt wird, abgelesen werden.

Dass ein Deadlock, wie in Abbildung 5.8 gezeigt, immer eine zirkuläre Abhängigkeit ist, lässt sich leicht erkennen: Die Abhängigkeiten der Locations bilden einen Graphen. Solange es im Graphen der Abhängigkeiten keinen Ring gibt, gibt es am Ende jedes Pfads im Graphen eine Location, die gerade kein Signal prüft, also nicht blockiert ist. Im Beispiel in Abbildung 5.10(a) ist das die orange ausgefüllte.

In einem Ring hingegen ist der Thread mit der höchsten PID derjenige, der bei einer rein sequentiellen Ausführung vor den anderen im Ring ausgeführt würde und

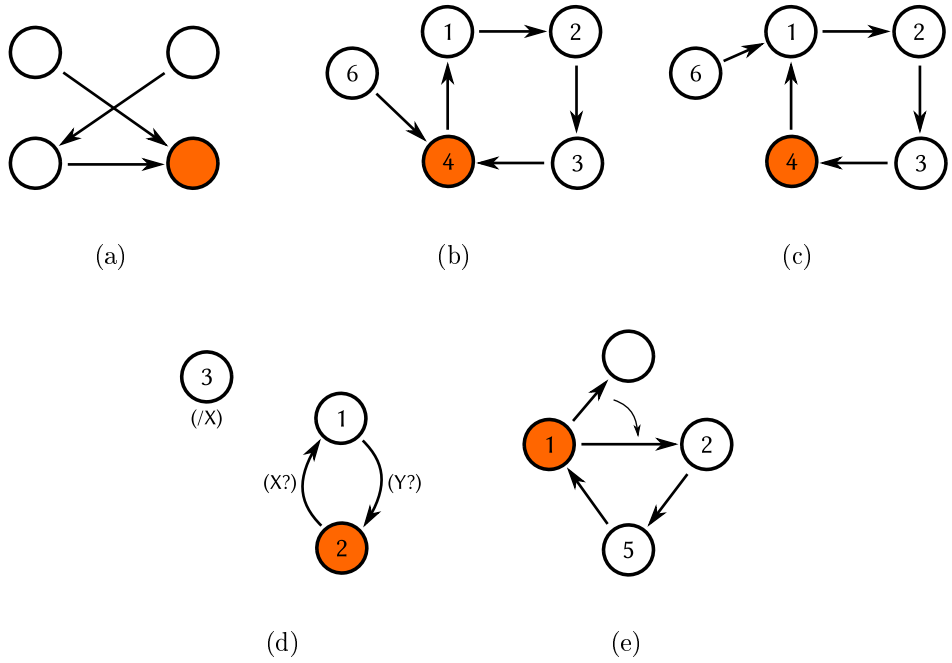


Abbildung 5.10.: Beispielgraphen zum LCR-Algorithmus

somit feststellen dürfte, dass das Signal, das er prüft, ABSENT ist. Im verteilten SCmc-Programm werden auf allen Locations die Threads mit den höchsten Prioritäten ausgeführt. Höherpriore Emitter innerhalb des Rings sind also für den Tick fertig, niederpriore dürfen das Signal im laufenden Tick nicht mehr senden. Da vom LCR-Algorithmus die größeren PIDs weitergeleitet werden, kommt bei einem Deadlock die PID des Threads mit der höchsten PID im Ring irgendwann bei seiner Location wieder an, und er weiß, dass es im Ring keine potentiellen Emitter mehr gibt.

Kann dazu noch ausgeschlossen werden, dass es höherpriore potentielle Emitter außerhalb des Rings, wie in Abbildung 5.10(d) zu sehen, gibt, so kann der Algorithmus im Fall $p_l = p_r$ sogar entscheiden, dass das geprüfte Signal ABSENT ist.

Für alle anderen Fälle ist dadurch, dass nur Annahmen über Locations im Ring gemacht werden, ein Fehler ausgeschlossen. Die orange ausgefüllte Location in 5.10(d) darf gefahrlos annehmen, dass von der Location mit PID 1 kein X mehr kommt, nicht aber von der Location mit der PID 3. Sie wird also als nächstes auf letztere warten.

Ist eine Location außerhalb des Rings, auf der ein Thread mit einer höheren PID als der aller anderen im Ring läuft, von einem Teil des Rings abhängig, so ist das unkritisch. Abbildung 5.10(b) zeigt so einen Fall. Die Location mit PID 6 wartet auf die orange ausgefüllte mit der PID 4. Beide müssen unterschiedliche Signale prüfen, denn die letztgenannte hat das Signal, das ihr aktiver Thread prüft, bereits lokal als „sicher“ markiert. Die erstgenannte würde nicht deswegen auf sie warten. Löst

sich der Deadlock auf, wird entweder die Abhängigkeit der Location außerhalb des Rings erfüllt oder es entsteht ein neuer Ring.

Threads, die dasselbe Signal prüfen, finden hingegen auch dieselbe `0` in der Certainty-Matrix, warten also auf die selbe Location. Ein solcher Fall ist in Abbildung 5.10(c) gezeigt. Auch dieser Fall ist unkritisch. Denn zum einen werden wie gesagt nur Annahmen über Locations im Ring gemacht. Zum anderen wird auch die Location mit dem höherprioren Prozess außerhalb des Rings das Signal, über das der höchstpriore Thread im Ring entscheidet, nicht mehr senden.

Um Störungen durch Veränderungen im Graphen auszuschließen, müssen alle empfangenen PIDs verworfen werden, sobald sich eine Abhängigkeit aufgelöst hat. Die in Abbildung 5.10(e) gezeigte Veränderung führt möglicherweise dazu, dass die orange gefüllte Location eine Nachricht mit der PID 5 verwirft, obwohl die Location mit PID 5 immer noch von ihr abhängig ist. Die orange gefüllte sendet aber bei der nächsten Blockade erneut ihre PID aus, so dass der Algorithmus im neu entstandenen Ring zum Erfolg führt.

Livelocks sind durch die Regel, dass eine `1` in der Certainty-Matrix nicht in eine `0` geändert werden darf, ausgeschlossen: Wenn eine Location auf eine andere wartet, dann nur auf diese eine, und nach Auflösung des Rings nicht wieder wegen desselben Signals auf dieselbe.

Der Algorithmus scheitert auch nicht an möglichen `FORK`s, mit denen höherpriore Prozesse erzeugt werden, weil Locations, die neue Threads abspalten, nicht blockieren, also auch nicht irrtümlich eine zu kleine PID weiterleiten.

Abgesehen von der ungeschickten Initialisierung der Certainty-Matrix mit `0`-Einträgen produziert der LCR-Algorithmus auch bei nichtkonstruktiven Programmen wie in Listings 5.11(a) und 5.11(b) eine Lösung, falls die beteiligten Threads über verschiedene Locations verteilt wurden und die Fehlererkennung von SC daher nicht funktioniert. Beide Signale würden im Zweifel als ABSENT angenommen.

```
1 T2:
2   if (PRESENT(A)) EMIT(B);
```

(a) Location 0

```
1 T1:
2   if (PRESENT(B)) EMIT(A);
```

(b) Location 1

Abbildung 5.11.: Nichtkonstruktives SCmc-Programm

Der LCR-Algorithmus kann durch Nachrichtenversand sicher feststellen, dass wirklich eine zirkuläre Abhängigkeit besteht. Zusätzliche Datenstrukturen im Shared Memory, die jeder Location eine Analyse der Situation gestatten, würden die Dynamik wechselnder Abhängigkeiten nicht berücksichtigen. Man könnte dort leicht die aktuell aktive PID und eine vermutete Blockade für andere Locations sichtbar anzeigen, doch wären Race-Conditions die Folge. Denn die Folge der aktiven PIDs ist wegen `FORK` und `PRIO` nicht monoton fallend, und die vermeintliche Blockade kann sich unbemerkt aufgelöst haben.

Die eindeutigen PIDs aus dem sequentiellen Programm beizubehalten, um die Auflösung von Deadlocks zu ermöglichen, wäre allerdings ebenfalls wenig sinnvoll. Die Certainty-Matrix würde durch unnötig hohe PIDs sehr groß.

Stehen dem LCR-Algorithmus keine eindeutigen PIDs zur Verfügung, so kann er die eindeutige Location-Nummer verwenden, um zirkuläre Abhängigkeiten zur Laufzeit zumindest zu erkennen. Die reine Erkennung löst letztendlich aber keinen Deadlock auf.

Will man die effiziente Neuvergabe von Prioritäten bei der Verteilung ermöglichen, aber die aus dem sequentiellen Programm trotzdem für LCR verwenden, hat man zwei Möglichkeiten: Man könnte die Threads anhand ihrer PIDs aufsteigend auf die Locations verteilen, so dass sich eine Priorisierung durch die Location-Nummer ergibt. Anhand dieser wäre dann eine Auflösung möglich. Dies dürfte aber nur in seltenen Fällen optimal sein.

Man könnte auch die PIDs als zusätzliches Datum speichern. Dabei müsste man aber auch alle PIDs des sequentiellen Programms kennen, die durch `PRIO`-Statements gesetzt werden, oder `PRIO` wirkungslos machen. Letzteres wäre eine zu starke Einschränkung. Denn ohne `PRIO` sind SyncCharts mit Makrozuständen, wie in Abbildung 5.12 gezeigt, nicht übersetzbar [3], falls die nebenläufigen Teile nicht auf verschiedene Locations verteilt werden.

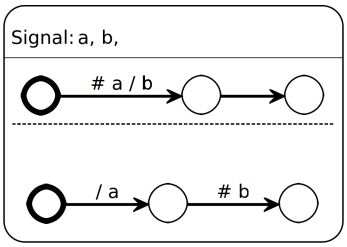


Abbildung 5.12.: SyncChart mit wechselnder Signalabhängigkeit (nach: Amende [3])

Für LCR benötigt man darüber hinaus die Möglichkeit, Nachrichten zwischen Locations auszutauschen und je nach Anzahl der Locations im System ausreichend große Empfangspuffer. Wie lange seine Ausführung dauert, ist unvorhersehbar.

Deadlock verhindern durch Initialisierungsvektoren

Eine weniger unvorhersehbare Methode wäre, zur Laufzeit auf jeder Location eine Initialisierungsmatrix zu pflegen, die vor jedem Tick in die Certainty-Matrix im Shared Memory kopiert wird. Beim Kompilieren wird für jeden Thread ein Initialisierungsvektor erzeugt, in dem bereits eingetragen ist, welche Signale er niemals sendet.

Als Optimierung könnte man sogar für jeden *Zustand* einen Vektor erzeugen, etwa das Ergebnis einer UND-Verknüpfung aller Vektoren der nächsten erreichbaren Zustände. Dieser wird dann zur Laufzeit vom jeweiligen Thread in `PAUSE` in die

```
1  T11:
2     FORK(ABE, 3);
3     FORK(CDE, 4);
4     FORKE(T12);
5
6  T12:
7     PAUSE;
8     EMIT(A);
9     ...
10
11 T21:
12    EMIT(B);
13    EMIT(C);
14    TERM;
15
16 ABE:
17   PAUSE;
18   if (...) EMIT(A);
19   if (...) EMIT(B);
20   ...
21
22 CDE:
23   AWAIT(A);
24   if (...) EMIT(C);
25   if (...) EMIT(D);
26   ...
```

Listing 5.10: abgespaltene Emitter mit initialem PAUSE

Initialisierungsmatrix kopiert. Da alle Veränderungen im lokalen Speicher der Location stattfinden, ist diese Methode relativ schnell im Vergleich zu allen, die eine Auflösung über den Shared Memory versuchen. Erst am Ende eines Ticks wird die lokale Initialisierungsmatrix dorthin übertragen.

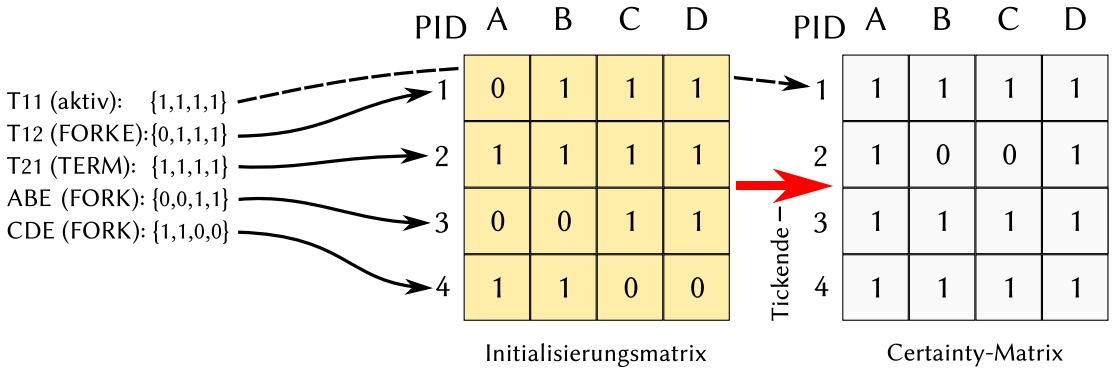


Abbildung 5.13.: Initialisierungsmatrix zu Listing 5.10

Allerdings müssen auch bei dieser Vorgehensweise unbenutzte Zeilen in der Certainty-Matrix besonders behandelt werden. Da wie zuvor erklärt FORKE nicht immer ausgeführt wird, gleichzeitig aber der einzig mögliche Ort für „Aufräumaktionen“ ist,

bleibt nur die Möglichkeit, FORK den Initialisierungsvektor für einen neu gestarteten Thread setzen zu lassen. Unbenutzte Zeilen werden in der Initialisierungsmatrix mit 1 gefüllt. PRIO vertauscht die Initvektoren der alten und neuen PIDs, und ABORT und TERM setzen den Initialisierungsvektor für beendete Threads auf 1 zurück.

Da der Vektor aber erst im nächsten Tick verwendet werden kann, muss jeder neue Thread einen Tick absolvieren, ohne Signale auszusenden. Er müsste also mit einem PAUSE oder einem Statement, das ein PAUSE enthält, beginnen.

Listing 5.10 enthält ein Beispiel. Die Zustände T11 und T21 seien mit den PIDs 1 und 2 die gerade aktiven. Abbildung 5.13 zeigt die Certainty-Matrix und wie die Initialisierungsmatrix verändert wird. Bei der Erzeugung von ABE und CDE mit FORK werden die Einträge für A und B beziehungsweise C und D in den Initialisierungsvektoren auf 0 gesetzt.

Da beide Threads zu Beginn ihrer Ausführung einen Tick pausieren, sind die aktuellen Einträge in der Certainty-Matrix korrekt. T11 springt mit FORKE in den Zustand T12, in dem A emittiert wird. Enthielte T12 kein initiales PAUSE, so hätte sein Initialisierungsvektor im letzten Tick anders gesetzt werden müssen. Bei neu erzeugten Zuständen geht dies aufgrund der Unvorhersehbarkeit nicht. Da T21 terminiert, setzt TERM seinen Vektor auf $(1, 1, 1, 1)$.

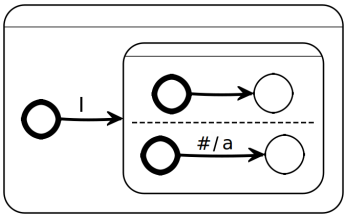


Abbildung 5.14.: SyncChart mit initialer Immediate-Transition

Auch wenn eine Initialisierungsmatrix sich mit verhältnismäßig geringem Aufwand pflegen lässt, ist der notwendige Verzicht auf Signalemissionen im ersten Tick neuer Threads hinderlich. Der SyncChart in Abbildung 5.14 ist zum Beispiel unübersetzbar.

5.3. Prioritätenbasierte Synchronisation

Bei der prioritätenbasierten Synchronisation wird zwischen den Locations abgestimmt, Threads welcher Priorität ausgeführt werden dürfen. Threads auf verschiedenen Locations dürfen die gleiche Priorität haben, wenn sie voneinander unabhängig sind. Die Prioritäten beziehungsweise PIDs haben zwar globale Bedeutung, müssen aber nur lokal eindeutig sein und nicht exakt denen im sequentiell ausgeführten Programm entsprechen.

Wenn die Prioritäten korrekt vergeben wurden, bleiben die Signalabhängigkeiten trotz paralleler Ausführung berücksichtigt. Jeder Thread weiß zu jedem Zeitpunkt, dass es im laufenden Tick keinen höherprioren Emitter eines Signals, dessen Status

er prüft, mehr gibt. Die normale Kodierung von Signalen kann also beibehalten werden. Ein Signal ist entweder PRESENT oder ABSENT, den Zustand $\perp$ gibt es nicht.

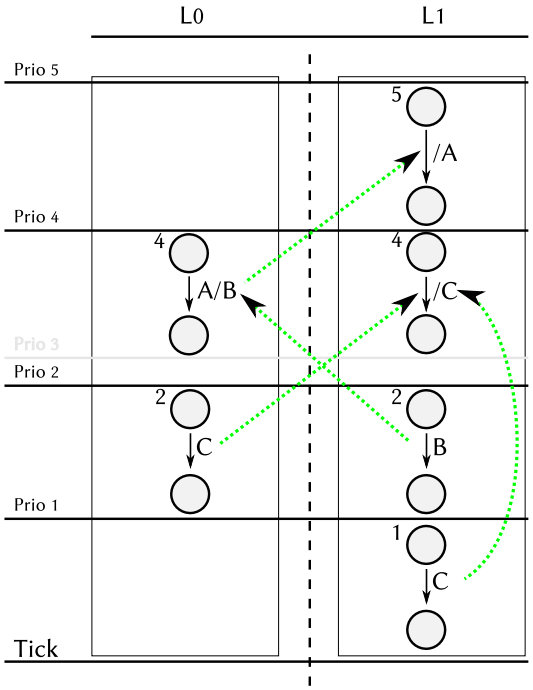


Abbildung 5.15.: Prioritätenbasierte Synchronisation

Abbildung 5.15 zeigt eine über Prioritäten synchronisierte Ausführung. Auf Location 1 wird ein Thread mit der Priorität 5 ausgeführt. Auf Location 0 gibt es keinen Thread mit Zuständen dieser Priorität. Erst wenn die Ausführung mit der Priorität 5 vollständig abgeschlossen ist, werden die Threads mit Priorität 4 ausgeführt. Der Thread auf Location 0 prüft das Signal `A`. Es wurde im Schritt zuvor ausgesendet, ist also PRESENT. Die Signalabhängigkeiten sind als grün gepunktete Pfeile eingezeichnet. Die Ausführung der Threads mit Priorität 3, hellgrau angedeutet, wird übersprungen, da es keine Threads mit dieser Priorität gibt. Im letzten Schritt, bei Priorität 1, wird wieder nur ein Thread ausgeführt, danach endet der Tick.

Listing 5.16(a) zeigt noch einmal das Standard-Beispiel ABRO, Listing 5.16(b) und 5.16(c) Code einer auf zwei Locations verteilten Version. Der einzige Unterschied zur Version für signalbasierte Synchronisation ist, dass die Prioritäten nicht möglichst direkt aufeinanderfolgend vergeben wurden. Lediglich den beiden unabhängigen Zuständen `WaitA` und `WaitB` wird dieselbe zugewiesen. Wird Code mit unveränderten Prioritäten verteilt, so entspricht die prioritätensynchrone Ausführung der sequentiellen mit etwas Overhead.

Da die Folge der Prioritäten nicht monoton fallend ist, ist ein einfaches Durchlaufen aller Prioritäten mittels eines globalen Zählers keine korrekte Lösung. Statt-

```
1  TICKSTART(4);
2
3  ABO:
4     FORK(AB, 1);
5     FORKE(ABOMain);
6
7  AB:
8     FORK(WaitA, 2);
9     FORK(WaitB, 3);
10    FORKE(ABMain);
11
12 WaitA:
13    AWAIT(A);
14    TERM;
15
16 WaitB:
17    AWAIT(B);
18    TERM;
19
20 ABMain:
21    JOIN;
22    EMIT(O);
23    TERM;
24
25 ABOMain:
26    AWAIT(R);
27    TRANS(ABO);
28
29 TICKEND;
```

(a) ABRO

```
1  TICKSTART(4);
2
3  ABO:
4     FORK(AB, 1);
5     FORKE(ABOMain);
6
7  AB:
8     FORK(WaitA, 3);
9     FORKE(ABMain);
10
11 WaitA:
12    AWAIT(A);
13    TERM;
14
15 ABMain:
16    JOIN;
17    EMIT(O);
18    TERM;
19
20 ABOMain:
21    AWAIT(R);
22    TRANS(ABO);
23
24 TICKEND;
```

(b) Location 0

```
1  TICKSTART(4);
2
3  ABO:
4     FORK(AB, 1);
5     FORKE(ABOMain);
6
7  AB:
8     FORK(WaitB, 3);
9     FORKE(ABMain);
10
11 WaitB:
12    AWAIT(B);
13    TERM;
14
15 ABMain:
16    JOIN;
17    EMIT(O);
18    TERM;
19
20 ABOMain:
21    AWAIT(R);
22    TRANS(ABO);
23
24 TICKEND;
```

(c) Location 1

Abbildung 5.16.: ABRO bei prioritätenbasierter Synchronisation

dessen müssen die Locations die Priorität der als nächstes auszuführenden Threads untereinander abstimmen. Jede Location sendet dazu zu Beginn eines Ticks die höchste aktive lokale Priorität an einen Koordinator. Dieser antwortet allen Locations mit dem Maximum der vorgeschlagenen Prioritäten.

Ein Vorteil dieser Methode ist, dass sie kein A-Priori-Wissen über die Prioritäten auf den verschiedenen Locations benötigt. Die Prioritäten müssen aber angepasst werden, um eine Parallelausführung zu bewirken.

Bei einem Wechsel der Priorität mit `PRIO` wird die Ausführung des Threads wie bei der sequentiellen Ausführung unterbrochen und auf die nächste Synchronisation gewartet. Die Makros `ABORT` und `JOIN` funktionieren lokal unverändert.

Die Priority-Barrier

Der Abgleich der Prioritäten erfolgt mittels einer erweiterten Barrier, der *Priority-Barrier*, bei der eine Location nicht nur anzeigt, dass sie sie erreicht hat, sondern auch ihre „Wunschpriorität“ bekanntgibt. Bei einer Shared-Memory-Umsetzung genügt dazu der Eintrag der Priorität in ein Array, auf dessen einzelne Werte nur je eine Location schreibenden Zugriff hat, und das vom Koordinator gelesen wird. Der Eintrag des Koordinators wird nur von diesem geschrieben und von allen anderen ge-

lesen. Wird die Barrier geöffnet, so enthält er die nächste zu bearbeitende Priorität. Eine Priority-Barrier ähnelt also einem Beta-Synchronizer [7], bei dem es ebenfalls einen zentralen Koordinator gibt.

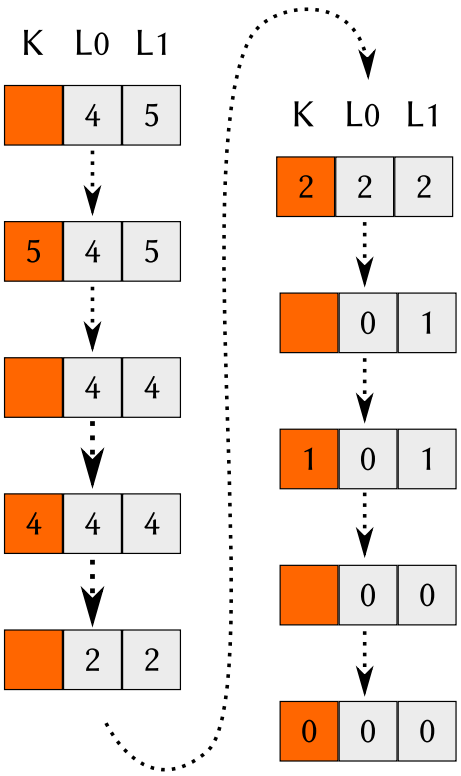


Abbildung 5.17.: Inhalt einer Priority-Barrier

Passend zum Beispiel in Abbildung 5.15 ist in Abbildung 5.17 der Inhalt einer Priority-Barrier vereinfacht gezeigt. Die Locations 0 und 1 nutzen die hellgrauen Felder, der Koordinator das orangefarbene, ganz linke.

Da sich die Information, dass eine Location eine Barrier erreicht hat, mit der, welche Priorität als nächste zur Ausführung ansteht, kombinieren lässt, eignet sich die prioritätenbasierte Synchronisation gut für Systeme, auf denen Message-Passing zur Verfügung steht. Der Empfang von Signalnachrichten kann außerdem, wenn gepufferte FIFO-Kanäle existieren, während der Synchronisation stattfinden. Signale, die im letzten Prioritätsschritt emittiert und als Nachricht gesendet wurden, können schließlich frühestens für den nächsten Prioritätsschritt relevant sein.

Das nichtdeterministische Beispiel in Listing 5.11 kann, unabhängig von der Priorität der Threads `T1` und `T2`, nicht als fehlerhaft erkannt werden, da es keine unmittelbare Wechselwirkung zwischen den Threads gibt. Haben sie unterschiedliche Priorität und ist das zuerst geprüfte Signal ABSENT, das zweite jedoch PRESENT, so wird ein auf einer anderen Location bereits geprüftes Signal emittiert, ohne dass dies bemerkt würde. Im Falle eines monolitischen Programms wäre von einem Compiler ein Abhängigkeitszyklus erkannt worden.

5.4. Hierarchieabhängigkeiten zwischen Locations

In den Abschnitten 5.3 und 5.2 wurden die Operatoren `FORK`, `ABORT` und `JOIN` nur bezüglich ihrer Wirkung auf der eigenen Location betrachtet, beziehungsweise eine irgendwie geartete Abstimmung des Kontrollflusses vorausgesetzt.

In den Beispielen 5.3 und 5.16 wurden Zustände, die diese Operatoren enthalten, einfach auf jeder Location repliziert. `FORK`-Statements, die Threads starten würden, welche einer anderen Location zugewiesen wurden, wurden entfernt.

Dass die nebenläufigen Zweige auch alle im selben Tick gestartet werden, ergibt sich bei diesen Beispielen von selbst. Dies sollte aber auch bei anderen Programmen möglich sein, und zwar ohne sämtliche Zustände zu replizieren, die den `FORK`-Statements vorausgehen. Jede Location soll nach Möglichkeit nur den Teil des Programms ausführen, der ihr zugewiesen wurde. Nicht parallelisierbare Teile zu vervielfältigen ist unnötig energieaufwändig und verhindert spätere Optimierungen, zum Beispiel der Signalzustellung. Würde ein SyncChart wie in Abbildung 5.14 übersetzt und verteilt, so müsste jede Location zunächst den initialen Zustand ausführen und das Signal `I` prüfen.

Ebenso muss die Beendigung eines Threads auf einer anderen Location erkannt werden können. Wird in `JOIN` keine entsprechende Funktionalität eingebaut, so führt die Verteilung schon beim Beispiel ABRO zu einem Fehler: Die `JOIN`-Statements in Zeile 16 in den Listings 5.16(b) und 5.16(c) erkennen jeweils nur die Beendigung eines Threads, das Signal `O` wird also schon gesendet, sobald nur `A` oder nur `B` PRESENT ist.

In die Thread-Tabellen anderer Locations hineinzusehen oder dort hineinzuschreiben ist nicht möglich. Entsprechende Datenstrukturen im Shared Memory würden wiederum eindeutige Kennungen und Schutz vor Race Conditions erfordern.

Stattdessen bietet es sich an, die Hierarchieabhängigkeiten mit Signalen aufzulösen. Dies könnte zukünftig mit eigens dafür eingerichteten System-Signalen realisiert werden. `FORK` und `JOIN` würden dann in SCmc genauso wie in SC funktionieren. Es kann aber auch einfach die vorhandene Funktionalität genutzt werden.

Die Listings in Abbildung 5.18 zeigen, wie die Locations einfach durch das Senden und Abwarten eines zusätzlichen *Termination-Signals* `T_WaitB` synchronisiert werden können. Auf Location 1 wurden das `EMIT(O)` in Zeile 17 aus `ABMain` entfernt und ein `EMIT(T_WaitB)` eingefügt. Auf Location 0 wurde dem `JOIN` in Zeile 17 ein `AWAIT(T_WaitB)` vorangestellt.

Das zusätzliche `AWAIT`-Statement prüft in jedem Tick, ob das Termination-Signal PRESENT ist. Ist dies der Fall, so setzt die Ausführung mit `JOIN` fort. Das Signal `O` kann also erst gesendet werden, wenn beide Child-Threads terminiert sind.

Die Synchronisation über zusätzliche Signale kann problemlos auf beliebig viele Threads erweitert werden. Die Listings in Abbildung 5.19 zeigen ein Beispielprogramm für drei Locations mit einer generischeren Ersatzfunktion für `JOIN`, `FJOIN` in Zeile 30. Die Termination-Signale sind `T2` und `T3`. Hier wird ein lokaler Zähler genutzt, um die Zahl der aktiven Child-Threads auf entfernten Locations zu speichern. Er wird nach jeder Erzeugung eines entfernten Threads inkrementiert.

```
1  TICKSTART(4);
2
3  ABO:
4     FORK(AB, 1);
5     FORKE(ABOMain);
6
7  AB:
8     FORK(WaitA, 3);
9     FORKE(ABMain);
10
11 WaitA:
12    AWAIT(A);
13    TERM;
14
15 ABMain:
16    AWAIT(T_WaitB);
17    JOIN;
18    EMIT(O);
19    TERM;
20
21 ABOMain:
22    AWAIT(R);
23    TRANS(ABO);
24
25 TICKEND;
```

(a) Location 0

```
1  TICKSTART(4);
2
3  ABO:
4     FORK(AB, 2);
5     FORKE(ABOMain);
6
7  AB:
8     FORK(WaitB, 3);
9     FORKE(ABMain);
10
11 WaitB:
12    AWAIT(B);
13    TERM;
14
15 ABMain:
16    JOIN;
17    EMIT(T_WaitB);
18    TERM;
19
20 ABOMain:
21    AWAIT(R);
22    TRANS(ABO);
23
24 TICKEND;
```

(b) Location 1

Abbildung 5.18.: ABRO mit ersetztem `JOIN`

Denkbar wäre hier auch die Nutzung von Valued Signals. `FORK`ende Threads würden ein Valued Signal mit `+1`, terminierende mit `-1` emittieren. Als Combine-Funktion würde eine Addition gewählt, so dass das Valued Signal wie ein Zähler wirkt.

Die neu entstehenden Signalabhängigkeiten führen dazu, dass bei prioritätenbasierter Synchronisation bislang gleichzeitig ausgeführte Zustände, im Beispiel 5.18 `AB` in Zeile 7 und `ABMain` in Zeile 15, nicht mehr gleichzeitig ausgeführt werden. Dies betrifft aber nur kurze Zustände, die Hierarchieabhängigkeiten abbilden.

Die Listings 5.19(a), 5.19(b) und 5.19(c) zeigen außerdem die Synchronisation von `FORK`-Statements. Die Situation bezüglich der Prioritäten ist hier genau umgekehrt: Die `FORK`enden Threads auf Locations außer 0 müssen bei prioritätenbasierter Synchronisation eine niedrigere Priorität haben als der auf Location 0. Das ist im Beispiel durch das neu eingefügte `PRIO` gewährleistet. Bei größeren Programmen wird eventuell eine Spreizung der Prioritäten notwendig, um Platz für die neuen Abhängigkeiten zu schaffen. Bei signalbasierter Synchronisation müssen die Prioritäten nicht angepasst werden.

Der Zustand `DELAY` in Zeile 5 konnte gekürzt werden. Er dient nur noch dazu, die Ausführung bei `FORKER` fortzusetzen, wo in Zeile 10 in Listings 5.19(b) und 5.19(c) auf die neu eingeführten Startsignale `F2` und `F3` gewartet wird.

Vergleichsweise simpel ist die Umsetzung einer Strong Abortion auf mehreren Locations: Wie `ABOMain` in Zeile 20 in Beispiel 5.18 wird der entsprechende Zustand

```
1  TICKSTART(4);
2
3  do {
4
5  DELAY:
6     PAUSE;
7     PAUSE;
8     PAUSE;
9     GOTO(FORKER);
10
11 FORKER:
12    FORK(F1, 2);
13    EMIT(F2);
14    Fchildren++;
15    EMIT(F3);
16    Fchildren++;
17    FORKE(FORKER2);
18
19 F1:
20    PAUSE;
21    PAUSE;
22    PAUSE;
23    TERM;
24
25 FORKER2:
26    PAUSE;
27    CALL(FJOIN);
28    TERM;
29
30 FJOIN:
31    PRIO(1);
32    while (Fchildren != 0) {
33       if (PRESENT(T2)) Fchildren--;
34       if (PRESENT(T3)) Fchildren--;
35       if (Fchildren != 0) PAUSE;
36    }
37    JOIN;
38    RET;
39
40 } while (1);
41
42 TICKEND;
```

(a) Location 0

```
1  TICKSTART(4);
2
3  do {
4
5  DELAY:
6     GOTO(FORKER);
7
8  FORKER:
9     PRIO(3);
10    AWAITI(F2);
11    FORK(F2, 2);
12    FORKE(FORKER2);
13
14 F2:
15    TERM;
16
17 FORKER2:
18    JOIN;
19    EMIT(T2);
20    TERM;
21
22 } while (1);
23
24 TICKEND;
```

(b) Location 1

```
1  TICKSTART(4);
2
3  do {
4
5  DELAY:
6     GOTO(FORKER);
7
8  FORKER:
9     PRIO(3);
10    AWAITI(F3);
11    FORK(F3, 2);
12    FORKE(FORKER2);
13
14 F3:
15    TERM;
16
17 FORKER2:
18    JOIN;
19    EMIT(T3);
20    TERM;
21
22 } while (1);
23
24 TICKEND;
```

(c) Location 2

Abbildung 5.19.: Ersetzte `FORK` und `JOIN`

des Parent-Threads auf alle Locations verteilt. Alle Locations prüfen also das Abbruchsignal, im Beispiel `R`, und brechen ihre Child-Threads mit `TRANS` oder `ABORT` ab, wenn es PRESENT ist. Weak Abortions funktionieren analog.

Ein Vorteil von SC ist, dass keine von jedem Thread auszuführenden Checkabort-Instruktionen wie bei PRET-C benötigt werden [62, S. 49]. Dies gilt bei SCmc nur noch bedingt, denn ein `ABORT` wirkt nur auf lokale Child-Threads. Da die Prüfung parallel ausgeführt werden kann, führt dies aber nicht zu Zeitverlusten.

Zustände, in denen Abbruchbedingungen oder Terminations geprüft werden, sollten natürlich minimal sein. Während der vorhandene SyncCharts-Compiler das sicherstellt, könnte handgeschriebener SCmc-Code aufwändige Berechnungen enthal-

ten. Das ist schlichtweg schlechte Programmierung. Abbruchbedingungen, die komplizierter als eine Signalabfrage sind, sollten nur an einer Stelle geprüft und dann ein Abbruchsignal gesendet werden.

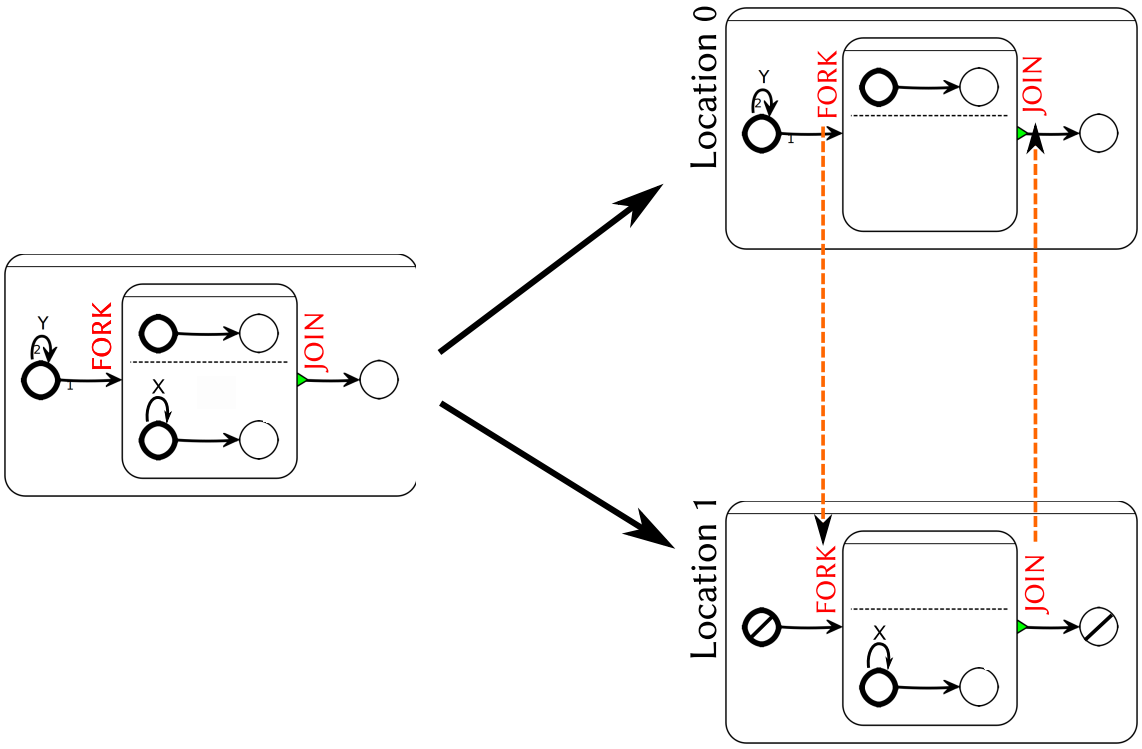


Abbildung 5.20.: Verteilung eines SyncCharts mit Hierarchie I

Abbildung 5.20 zeigt die Verteilung eines einfachen Programms in SyncCharts. Das Ausgangsprogramm ist ein SyncChart mit einem Makrozustand, welcher zwei parallele Bereiche enthält. Durch die Signale `X` und `Y` hängt es von Eingabesignalen ab, wann die beiden Bereiche ausgeführt werden und wann der untere terminiert. Location 0 soll den oberen Bereich aus dem Makrozustand ausführen, Location 1 den unteren. Beiden Locations müssen je ein `FORK` und ein `JOIN` ausführen, die Kommunikation zur Synchronisation ist orange gestrichelt eingezeichnet.

Die durchgestrichenen Zustände auf Location 1 können auf `GOTO`-Sprünge reduziert werden. Die Gefahr, dass `PAUSE`-Statements, die der Synchronisierung nebenläufiger Threads dienen, verloren gehen, besteht nicht. Nur replizierte Zustände auf sekundären, das heißt für die jeweilige Hierarchieebene nicht den Kontrollfluss bestimmenden Locations werden reduziert.

5.5. Vergleich signalbasierter und prioritätenbasierter Synchronisation

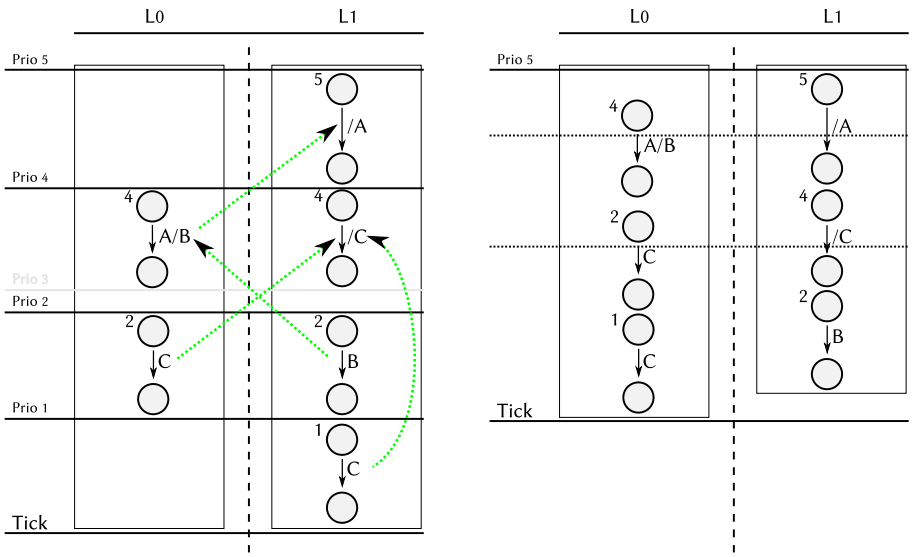
Die signalbasierte Synchronisation ignoriert eine Grundidee synchroner Programmierung, indem sie den Zustand $\perp$ einführt. Die zusätzlich notwendige Certainty-Matrix bringt einige Probleme mit sich. Eine Nullinitialisierung ist in den meisten Fällen unmöglich, weil dadurch Deadlocks entstehen können. Deadlocks zu erkennen und aufzulösen ist aufwändig und führt zu unvorhersehbarem Zeitverhalten und Einschränkungen. Initialisierungsvektoren lassen nur noch eine Teilmenge von SyncCharts zur Übersetzung zu oder verzögern bei angepassten SyncCharts die Ausführung neuer Threads um einen Tick. Das sind starke Einschränkungen, die die signalbasierte Synchronisation unattraktiv erscheinen lassen. Andererseits bietet sie eine gewisse Flexibilität bei der parallelen Ausführung.

Die prioritätenbasierte Synchronisation ist offensichtlich simpel und absehbar einfach zu implementieren. Sie benötigt keine Datenstruktur, deren Größe von der Zahl der Signale oder der der Threads abhängt. Prioritäten können also mit Lücken vergeben werden. Negative Wechselwirkungen zwischen den Locations gibt es nicht, und es müssen keine besonderen Mechanismen – bis auf die Priority-Barrier – eingesetzt werden. Einschränkungen wie unübersetzbare Klassen von SyncCharts gibt es ebenfalls nicht.

Bezüglich der im vorigen Abschnitt beschriebenen Synchronisation von `FORK` und `JOIN` unterscheiden sich die Methoden nur minimal. Die prioritätenbasierte Methode erfordert eine sequentielle Ausführung der verteilten Zustände, doch das dürfte praktisch keine Auswirkungen auf die Laufzeit haben, wenn diese nicht überfrachtet wurden. Damit Programmteile parallel ausgeführt werden, müssen die Prioritäten der unabhängigen Teile identisch sein. Bei der signalbasierten Synchronisation ist das nicht nötig, die Threads können hier praktisch beliebig verteilt werden.

In Abbildung 5.21 ist noch einmal die Ausführung aus Abschnitt 5.3 mit den grün gestrichelten Signalabhängigkeiten gezeigt, daneben zum Vergleich eine theoretische optimale Ausführung desselben Programms bei signalbasierter Synchronisation. Der Übersichtlichkeit halber wurden Unterbrechungen und Verzögerungen nicht eingezeichnet und stattdessen die Synchronisation über die Signale durch gepunktete Linien angedeutet. Wie man sieht, könnte die signalbasierte Synchronisation die Locations besser ausnutzen, wenn ein Signalzustand frühzeitig bekannt ist. Die Priority-Barriers verhindern dies.

Abbildung 5.22 zeigt den Fall, dass nicht die Priority-Barriers, sondern länger rechnende Zustände, Long-Duration-Tasks (LDTs), die Ausführung verzögern. Sie sind dunkelgrau und langgezogen dargestellt. Wegen der Abhängigkeit von Signal `A` kann der LDT auf Location 0 nicht gleichzeitig mit dem auf Location 1 ausgeführt werden. Er zieht dadurch die Zeit zwischen dem Erreichen von Priorität 4 und Priorität 2 in die Länge. Die beiden Threads mit Priorität 2 auf Location 1 und 2 könnten deutlich früher ausgeführt werden.



(a) Prioritätenbasierte Synchronisation (b) Signalbasierte Synchronisation

Abbildung 5.21.: Vergleich verschieden synchronisierter Ausführungen

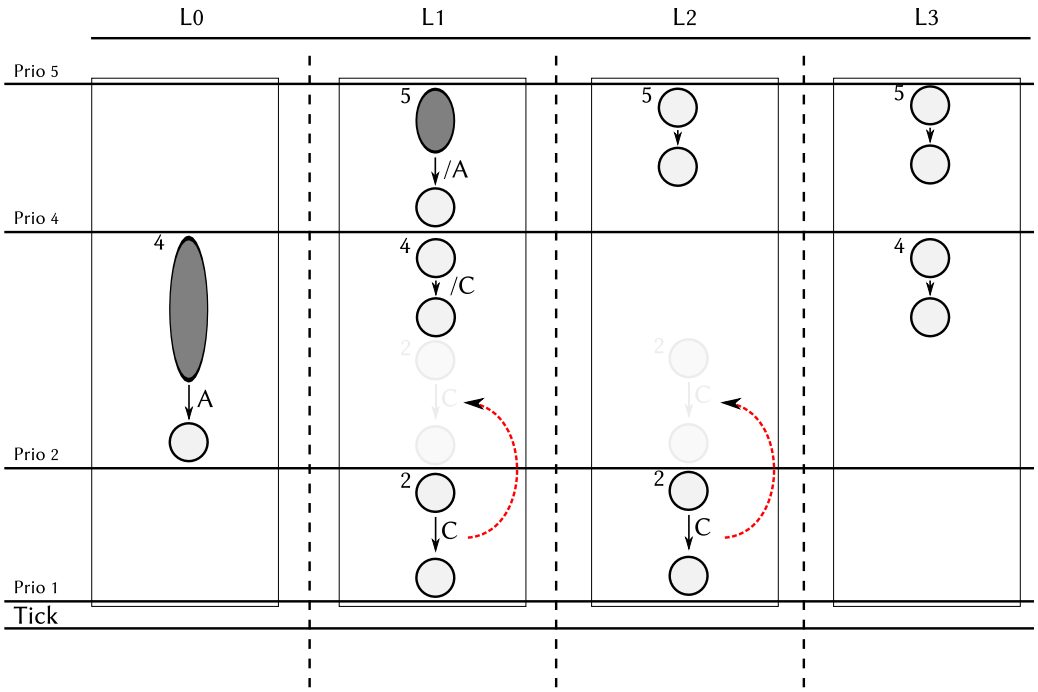


Abbildung 5.22.: Zeitverlust durch LDTs bei prioritätenbasierter Synchronisation

Ob ein solcher Fall eintritt, hängt letztendlich aber vom Programm und der Granularität der Threads und Zustände ab. Auch wenn lange rechnende Zustände im

Vorfeld erkannt und in kleinere Teile zerlegt werden, hängt die Ausführungszeit eines Prioritätsschrittes immer vom langsamsten Thread ab.

5.6. Ansätze zur Synthese parallelen Codes

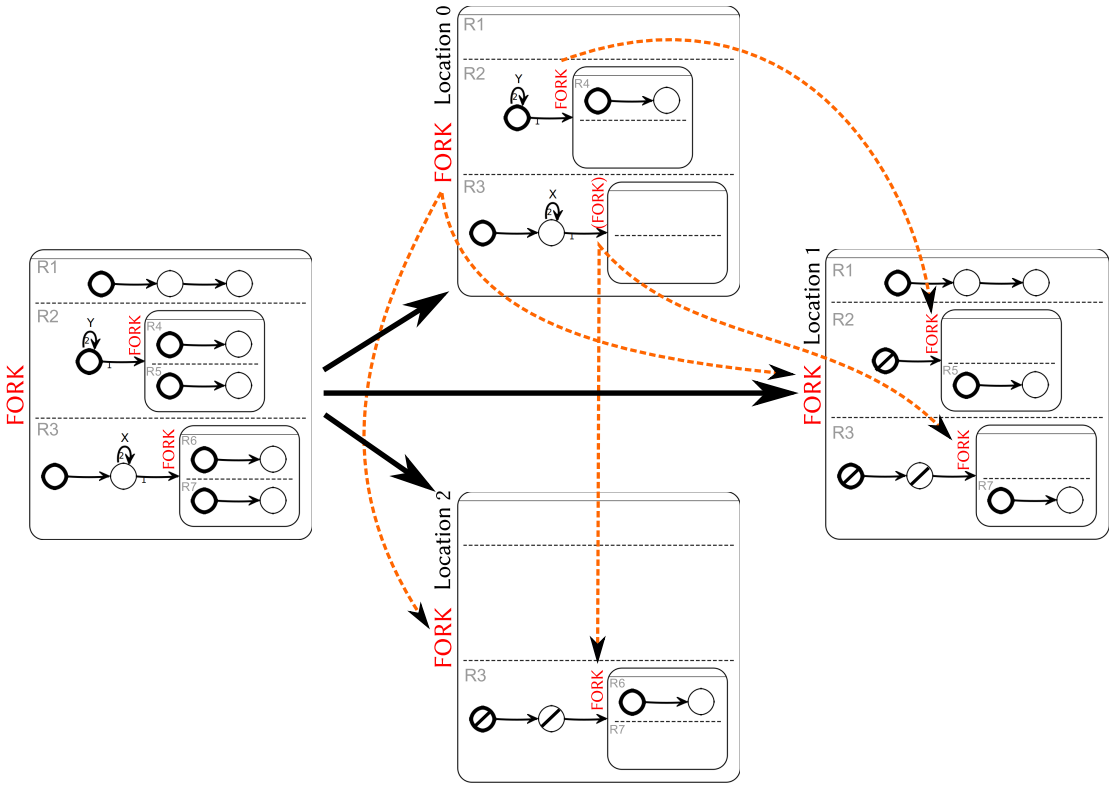


Abbildung 5.23.: Verteilung eines SyncCharts mit Hierarchie II

Abbildung 5.23 zeigt eine mögliche Verteilung eines SyncCharts mit mehrfachen parallelen Konstrukten. `FORK`-Statements, die im synthetisierten SC beziehungsweise SCmc-Code stehen, sind in den SyncChart eingezeichnet. Im Gegensatz zum Beispiel in Abbildung 5.20 müssen hier nur `FORK`-Statements synchronisiert werden, diese aber zu ganz unterschiedlichen, von den Signalen `X` und `Y` abhängigen Zeitpunkten. Startsignale, wie in Abschnitt 5.4 beschrieben, sind daher notwendig. Sie sind als orange gestrichelte Pfeile eingezeichnet.

Eine Verteilung könnte so ablaufen, dass der Anwender oder ein Programm Regionen aus Makrozuständen verschiedenen Locations zuweist. Region R1 wird im Beispiel Location 1 zugewiesen, ebenso R5 und R7. R2 wird nicht explizit zugewiesen. R6 soll auf Location 2 ausgeführt werden. Location 0 soll R4 und R3 ausführen. Da R2 R5 beinhaltet und R3 R6 und R7, erscheint dies zunächst widersprüchlich. Eine entsprechende Zuweisung ist aber möglich, wenn zunächst die äußeren Regionen zugewiesen wurden und im Anschluss zur Verfeinerung die inneren.

Der Compiler könnte nun zunächst das gesamte Programm auf alle Locations replizieren. Für die innersten Regionen R4 bis R7 sowie für R1 entfernt er dann die enthaltenen Zustände auf allen Locations, denen sie nicht zugewiesen wurden. Makrozustände, die diese Regionen enthalten, bleiben zunächst überall bestehen. Die anderen Zustände der nächsthöheren Hierarchieebene, also die in R2 und R3, werden im SCmc-Code auf `goto`-Sprünge reduziert, damit die `FORK`-Statements erreicht werden. Diese müssen um das Warten auf Startsignale ergänzt werden. Eine Ausnahme bilden die auf Location 0, denn R3 wurde dieser Location abweichend vom Inhalt des Makrozustands zugewiesen.

Die Zustände in R3 bleiben also erhalten, darunter auch das `FORK`, das allerdings zu einem reinen Sender von Startsignalen degeneriert. Die Startsignale sind an die korrespondierenden „empfangenden“ `FORK`s auf Location 1 und 2 gerichtet. Wenn die Startsignale PRESENT sind, starten diese sofort die Ausführung der Regionen R6 und R7.

R2 wurde nicht explizit zugewiesen, die enthaltenen Zustände werden daher auf einer der beiden Locations erhalten, die R4 und R5 enthalten. Im Beispiel ist das ebenfalls Location 0. Auf Location 1 wird der dem Makrozustand vorausgehende einfache Zustand reduziert.

Dieses Verfahren kann man prinzipiell für beliebig viele Hierarchieebenen fortsetzen. Im Beispiel führt es dazu, dass jeder einfache Zustand auf genau einer Location ausgeführt wird.

Bei der Reduktion entfallen auch Abhängigkeiten von äußeren Signalen. Die reduzierten Zustände dienen lediglich dazu, eventuelle `FORK`- oder `JOIN`-Statements zu erreichen, die dann auf Startsignale warten oder Termination-Signals senden. Kommt irgendwo ein `ABORT` vor, so muss beachtet werden, unter welcher Bedingung es ausgeführt wird. Einfache `AWAIT`-`ABORT`-Konstrukte können repliziert werden.

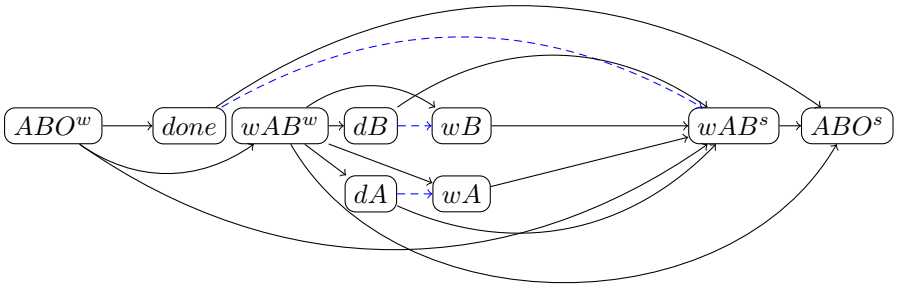


Abbildung 5.24.: Parallel ausführbare Zweige im geordneten Abhängigkeitsbaum

Durch die FORK/JOIN-Methodik von SC bleiben parallele Zweige des SyncCharts nach der Übersetzung mit dem Compiler von Amende [3] zwar als solche erkennbar. Der Compiler entfernt aber die Information, welche Threads im SyncChart tatsächlich unabhängig voneinander sind.

Für die prioritätenbasierte Synchronisation wäre es erforderlich, diese Information zu nutzen, um unabhängige Threads mit gleichen Prioritäten auszustatten. Dazu müsste die in Abschnitt 3.4 beschriebene Erzeugung der topologischen Ordnung modifiziert werden. Anstatt dem Abhängigkeitsbaum *irgendeinen* Wurzelknoten zu entnehmen, könnten *alle* gleichzeitig existierenden Wurzelknoten entnommen werden und die gleiche Priorität zugewiesen bekommen. Um Überpartitionierung zu verhindern, könnten in jedem Schritt nur so viele unabhängige Knoten gleiche Prioritäten zugewiesen bekommen, wie Locations vorhanden sind. Sind alle Locations besetzt, wird mit der nächsten Priorität fortgesetzt.

So entstehen im Prinzip mehrere parallele Ordnungen. Abbildung 5.24 zeigt das Ergebnis für den vereinfachten Abhängigkeitsbaum von ABRO. Man könnte einen topologisch geordneten Baum für jede Location anlegen und unabhängige Knoten nur je einem dieser Bäume zuordnen, alle anderen Knoten jedoch zunächst überall replizieren.

5.7. $\rightarrow_\beta$-Ordnung

Dass unabhängige Ereignisse innerhalb eines Ticks in beliebiger Reihenfolge auftreten dürfen, lässt sich auch formal zeigen. Die auf den einzelnen Kernen ausgeführten Programme entsprechen I/O-Automaten, wie von Lynch [45] beschrieben. Die einzelnen Threads entsprechen den „Tasks“ der Automaten [45, 14.1.1]. Als *Aktionen* des Automaten werden Transitionen sowie Empfang und Senden von Nachrichten, das heißt Signalen, bezeichnet. Eine *Ausführung* ist eine Folge von Zuständen und Aktionen eines Automaten.

Fair ist die Ausführung vereinfacht gesagt dann, wenn jede Aktion, so lange sie ausgeführt werden darf, auch unendlich oft die Chance dazu bekommt. Durch das prioritätenbasierte Scheduling ist das Fairnesskriterium hier automatisch erfüllt, es wird daher im Folgenden zugunsten der Lesbarkeit weggelassen. Da nur Abläufe innerhalb eines Ticks betrachtet werden, sind die Ausführungen zudem endlich. Ein *Trace* einer Ausführung ist die Folge der äußerlich wahrnehmbaren Ereignisse, das heißt der Sende- und Empfangsaktionen oder Sende- und Empfangs*ereignisse*. Wird von einem „Trace eines Automaten“ gesprochen, so ist der Trace einer Ausführung des Automaten gemeint.

Werden nun mehrere solche Automaten zusammengesetzt, so entsteht ein *asynchrones Broadcast-System* [45, 14.1.3], das heißt ein Automat, der aus Teilautomaten P_i zusammengesetzt ist, die wiederum über einen universellen, zuverlässigen Broadcast-Kanal verbunden sind. Abbildung 5.25 zeigt ein solches System mit drei Teilautomaten und einem zuverlässigen Broadcast-Kanal. Im Bild sendet Automat P_2 ein Signal S, welches vom Broadcast-Kanal an alle Automaten, auch P_2, verteilt wird. Offensichtlich entspricht die Ausführung auf verschiedenen Prozessorkernen einem solchen System.

Wie bei Lynch [45, 14.2.4] definiert man eine irreflexive partielle Ordnung:

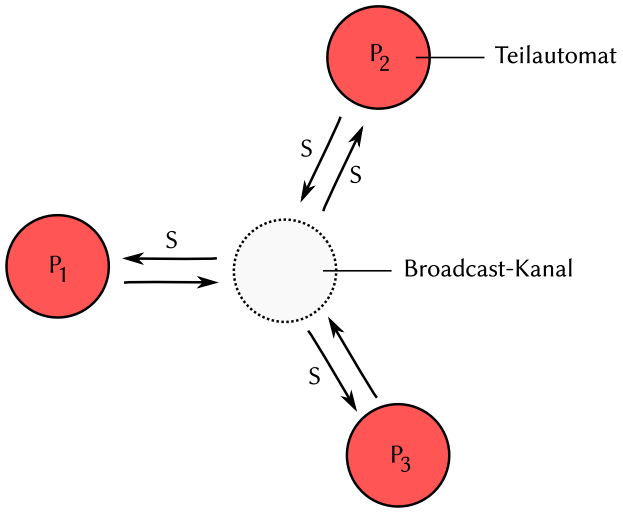


Abbildung 5.25.: Asynchrones Broadcast-System

Definition ($\rightarrow_\beta$-Ordnung, übersetzt & vereinfacht)**.** *Sei* A *ein asynchrones Broadcast-System. Sei* β *ein beliebiger Trace von* A*. Wenn* π *und* ϕ *zwei Ereignisse in* β *sind, wobei* π ϕ *vorausgeht, dann schreibt man:* $\pi \rightarrow_\beta \phi$ *oder man sagt: „*ϕ *hängt ab von* π*“, wenn einer der folgenden Punkte zutrifft:*

1. π *und* ϕ *sind Ereignisse desselben Prozesses* P_i*.*
2. π *ist eine Sendeaktion und* ϕ *ist die korrespondierende Empfangsaktion.*
3. π *und* ϕ *sind durch eine Reihe von Abhängigkeiten wie in 1. und 2. verbunden.*

Ununterscheidbarkeit [45, 8.7] ist so definiert:

Definition (Ununterscheidbarkeit, übersetzt)**.** *Ununterscheidbar sind zwei Ausführungen* α *und* α' *für einen Teilautomaten* P_i*, wenn die Folge der Ereignisse des Teilautomaten* P_i *in* α *mit der in* α' *übereinstimmt.*

Korollar 14.4 [45, 14.2.4] lautet nun:

Definition (Korollar 14.4, übersetzt & vereinfacht)**.** *Sei* A *ein asynchrones Broadcast-System. Sei* α *eine Ausführung von* A *und* β *der zugehörige Trace. Sei* γ *eine Folge, die durch Neuordnung der Ereignisse in* β *entstanden ist, wobei die* $\rightarrow_\beta$*-Ordnung erhalten wurde. Dann gibt es eine Ausführung* α' *von* A*, deren Trace* γ *ist, und die für jeden Teilautomaten* P_i *ununterscheidbar von* α *ist.*

Ereignisse ohne $\rightarrow_\beta$-Ordnung dürfen folglich in beliebiger Reihenfolge auftreten, das heißt auf mehreren Kernen parallel abgearbeitet werden, ohne dass dies für die Programme auf den einzelnen Kernen erkennbar wäre – ihr Verhalten bleibt

gleich. Wenn nun das Verhalten der einzelnen Teile gleich bleibt, dann hat jeder Teilautomat dieselben Signale gesehen und dieselben Signale gesendet. Außerdem muss jeder Thread aufgrund des deterministischen Verhaltens die gleichen Transitionen genommen haben. Also ist das Ergebnis des Gesamtsystems für diesen Tick ebenfalls identisch.

Eine Besonderheit des zuverlässigen Broadcast-Kanals bei Lynch ist, dass dieser eine FIFO-Ordnung zwischen den Nachrichten je zweier Kanäle erhält. Bei einer der in dieser Arbeit vorgestellten Implementierungen ist genau das der Fall. Das auf Kern 0 ausgeführte Programm übernimmt abwechselnd die Aufgabe des Broadcast-Kanals und die eines normalen Teilautomaten, dies ändert aber nichts an der Gültigkeit des oben Gesagten. Die Signale werden von den Teilautomaten alle zu Beginn eines Prioritätsschrittes empfangen und lokal vorgemerkt.

Die Kommunikation über Shared-Memory scheint hingegen nicht ganz dem Modell des asynchronen Send-/Receive-Systems von Lynch zu entsprechen. Schließlich entspricht jedes `EMIT` einer Sendeaktion, jedes Signal steht den anderen Teilautomaten instantan zur Verfügung, und diese können alle Signale in beliebiger Reihenfolge prüfen.

Das Shared-Memory-Modell lässt sich aber in einen äquivalenten Automaten abbilden, der für jedes Signal einen eigenen Teilautomaten enthält, welcher ein Signal empfängt und über verlässliche FIFO-Kanäle an alle anderen weiterleitet. Der Signal-Teilautomat empfängt diese Nachrichten oder Signale beliebig oft und sendet sie so oft an alle Teilautomaten weiter, wie diese es empfangen.

Für dieses System gelten das allgemeinere Theorem 14.1 und Korollar 14.2 [45, 14.1.4], mit den gleichen Schlussfolgerungen wie für Broadcast-Systeme. Jedes `EMIT` kann also als Sendeereignis und jedes `PRESENT` als Empfangsereignis betrachtet werden. Ebenso kann man Einträge in der Certainty-Matrix als Sendeereignisse auffassen.

Strenggenommen sind verschiedene Ausführungen hier für die Signal-Teilautomaten unterscheidbar, der reale Shared-Memory speichert allerdings keine Information darüber, wer ein Signal zuerst gesendet hat, so dass sein Verhalten trotzdem gleich bleibt.

In Abbildung 5.26 sind zwei ununterscheidbare Traces α und γ eines aus Teilautomaten P_1 (blau bzw. dunkelgrau) und P_2 (orange bzw. hellgrau) bestehenden Systems sowie die partiellen Abhängigkeiten ($\rightarrow_\beta$) zwischen Sende- und Empfangsereignissen gezeigt. „/A“ steht für ein `EMIT(A)` und „A?“ für ein `PRESENT(A)`.

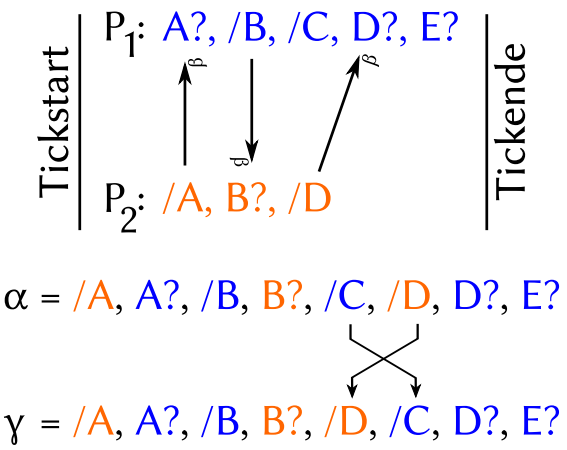


Abbildung 5.26.: Zwei ununterscheidbare Traces

6. Implementierungen und verwendete Technologien

Dieses Kapitel stellt die verwendeten Werkzeuge und Technologien vor, mit denen SCmc implementiert wurde, darunter den PRET-Simulator in Abschnitt 6.4. Abschnitt 6.5 erläutert kurz die Codestruktur von SC und SCmc. In den Abschnitten 6.6 und 6.7 werden die Implementierungen signalbasierter und prioritätenbasierter Synchronisation auf dem PRET und die Simulation verschiedener Locations mit Message-Passing unter Linux vorgestellt. Im Anschluss werden diese in Abschnitt 6.9 verglichen, und in Abschnitt 6.10 mögliche Optimierungen genannt. Abschnitt 5.6 enthält erste Ansätze zur Modifikation des SC-Compilers.

6.1. Compiler und Profiler

SC besteht fast ausschließlich aus Makro-Definitionen. An diesen scheitert das Syntax-Highlighting der üblichen Entwicklungsumgebungen. Dies erschwert die Orientierung erheblich, wenn Modifikationen vorgenommen werden sollen. Ein hohes Maß an Abstraktion führt dazu, dass eine Makroexpansion sich über zahlreiche Zwischenschritte erstrecken kann – die Makros sind ineinander verschachtelt. Listing 6.1 zeigt am Beispiel von `PRESENT(s)` die Makroexpansion über mehrere manuell erstellte Zwischenschritte.

Fehlermeldungen, die auf eine Zeile mit einem SC-Operator verweisen, sind wenig aussagekräftig, da sie die fehlerhafte Stelle in der Makro-Definition nicht erkennen lassen. Daher ist es nützlich, den C-Code mit expandierten Makros betrachten zu können. Dieser wird während des Kompilierens in temporären Dateien abgelegt. Deren Löschung kann aber verhindert werden, indem Compiler der GNU Compiler Collection (GCC) mit dem Parameter `-save-temps` aufgerufen werden. So kann darin nach der fehlerhaften Zeile gesucht oder eine expandierte Zeile in das ursprüngliche Programm kopiert und zur Lokalisierung des Fehlers um Umbrüche ergänzt werden. Dateien mit expandiertem C-Code haben die Endung `.i`. Listing 6.2 enthält das vom Compiler in eine Zeile expandierte `FORK(WaitA, 2)`, der ersten Zeile im Zustand `AB` aus ABRO in Listing 3.2.

Mit dem Parameter `-pg` aufgerufen erzeugt `gcc` eine Version des Programms, die zur Laufzeit Daten über Funktionsaufrufe generiert. Diese können mit einem Profiler wie `gprof` [31] oder einem grafischen Tool wie `KProf` ausgewertet werden, um das Verhalten des Programms zu analysieren. Statistische Daten über die Zahl der Funktionsaufrufe, die aufrufende Funktion und die Zeit, die in der jeweiligen

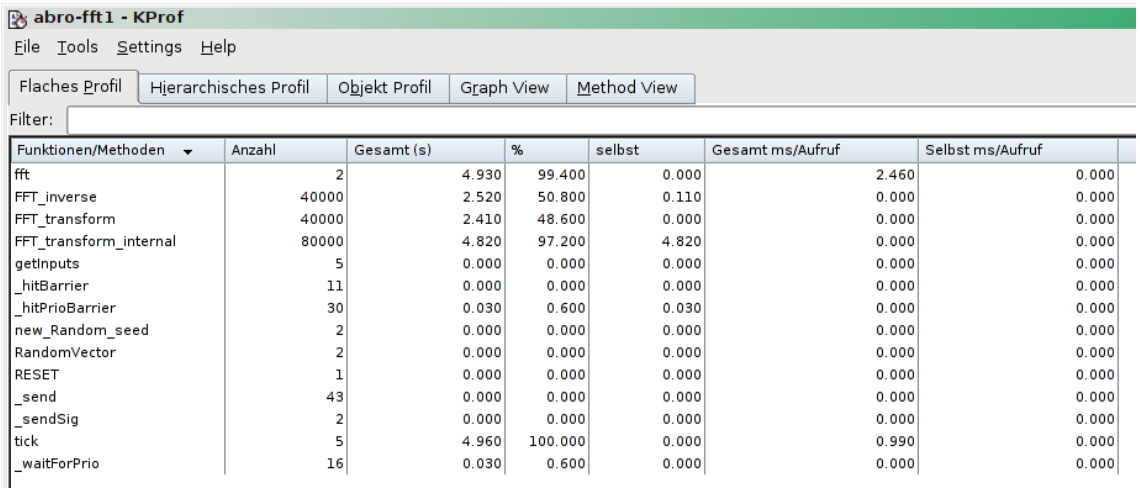
```
PRESENT(s)

(trace3tc("PRESENT:", "determines %s/%d %s\n", s2signame[s], s,
   _sigContains(signals, s) ? "present" : "absent")
   (_checkPRESENTc(s) _sigContains(signals, s)))

(traceThreadc("PRESENT:") trace3c("determines %s/%d %s\n",
   s2signame[s], s, _sigContains(signals, s) ?
   "present" : "absent")
   (_checkPRESENTc(s) _sigContains(signals, s)))

(instrCntIncrc trace3c("%-9s %d/%s ", "PRESENT:", _cid, state[_cid])
   trace3c("determines %s/%d %s\n", s2signame[s], s,
   _sigContains(signals, s) ?
   "present" : "absent") (_checkSIG_ID(s, _SC_SIG_MAX),
   (_sigContains(signals, s) ? 0 : _sigAdd(_presence_tested, s)),
   _setContains(signals, s)))

(tickInstrCnt++, printf("%-9s %d/%s ", "PRESENT:", _cid, state[_cid]),
   printf("determines %s/%d %s\n", s2signame[s], s,
   _sigContains(signals, s) ?
   "present" : "absent"), (_checkSIG_ID(s, _SC_SIG_MAX),
   (_sigContains(signals, s) ? 0 : _sigAdd(_presence_tested, s)),
   _setContains(signals, s)))

(tickInstrCnt++, printf("%-9s %d/%s ", "PRESENT:", _cid, state[_cid]),
   printf("determines %s/%d %s\n", s2signame[s], s,
   _setContains(signals, s) ?
   "present" : "absent"), (_checkSIG_ID(s, _SC_SIG_MAX),
   (_setContains(signals, s) ? 0 : _setAdd(_presence_tested, s)),
   _setContains(signals, s)))

(tickInstrCnt++, printf("%-9s %d/%s ", "PRESENT:", _cid, state[_cid]),
   printf("determines %s/%d %s\n", s2signame[s], s, (signals & u2b(s)) ?
   "present" : "absent"), (_checkSIG_ID(s, _SC_SIG_MAX),
   (signals & u2b(s)) ? 0 : (_presence_tested |= u2b(s))),
   (signals & u2b(s))))
```

Listing 6.1: Makroexpansion am Beispiel von `PRESENT(s)`

```
do { _pc[2] = &&WaitA; statePrev[2] = "_L_INIT"; state[2] = "WaitA"; _parent[2] = _cid; (_forkdescs |= (1
     << (2))); (enabled |= (1 << (2))); (active |= (1 << (2))); tickInstrCnt++; printf("%-9s %d/%s ", "
     FORK:", _cid, state[_cid]); printf("forks %d/%s, active = %s\n", 2, "WaitA", _set2str(_setstr, 4, (
     void *) &active)); } while (0);
```

Listing 6.2: Expandierte Version von `FORK(WaitA, 2)`

Funktion verbracht wurde, helfen insbesondere, Programmierfehler wie unerwünschtes Busy Waiting zu identifizieren. Abbildung 6.1 zeigt die Oberfläche von `KProf` mit der Aufrufstatistik eines Programms, dass etwa fünf Sekunden mit Fourier-Transformationen verbracht hat.



Funktionen/Methoden	Anzahl	Gesamt (s)	%	selbst	Gesamt ms/Aufruf	Selbst ms/Aufruf
fft	2	4.930	99.400	0.000	2.460	0.000
FFT_inverse	40000	2.520	50.800	0.110	0.000	0.000
FFT_transform	40000	2.410	48.600	0.000	0.000	0.000
FFT_transform_internal	80000	4.820	97.200	4.820	0.000	0.000
getInputs	5	0.000	0.000	0.000	0.000	0.000
_hitBarrier	11	0.000	0.000	0.000	0.000	0.000
_hitPrioBarrier	30	0.030	0.600	0.030	0.000	0.000
new_Random_seed	2	0.000	0.000	0.000	0.000	0.000
RandomVector	2	0.000	0.000	0.000	0.000	0.000
RESET	1	0.000	0.000	0.000	0.000	0.000
_send	43	0.000	0.000	0.000	0.000	0.000
_sendSig	2	0.000	0.000	0.000	0.000	0.000
tick	5	4.960	100.000	0.000	0.990	0.000
_waitForPrio	16	0.030	0.600	0.000	0.000	0.000

Abbildung 6.1.: KProf-Oberfläche

6.2. SciMark 2.0

Das Benchmark *SciMark 2.0* des National Institute of Standards and Technology (NIST) dient eigentlich dem Leistungsvergleich zwischen verschiedenen Prozessoren[1]. Es führt verschiedene mathematische Berechnungen mit Zufallszahlen durch, um anhand der Rechenzeit die Leistung des Prozessors in Million Floating Point Operations per Second (MFLOPS) zu bestimmen. Mittels eines Kommandozeilenparameters kann die Datenmenge vergrößert werden, um zu verhindern, dass die Felder vollständig in den Cache passen. SciMark 2.0 ist singlethreaded, es wird also nur die Rechenleistung eines Kerns bestimmt.

Da das NIST eine Regierungsbehörde der USA ist, ist der Quellcode von SciMark 2.0 gemeinfrei[2].

Aus SciMark 2.0 wurde C-Code für eine Fourier-Transformation verwendet, um damit künstlich Rechenlast in Zuständen erzeugen zu können.

6.3. Unix Domain Sockets

POSIX local inter-process communication sockets, auch *Unix Domain Socket* genannt[3], erlauben bidirektionale Kommunikation zwischen Prozessen desselben Hosts [56, S. 367ff]. Die Daten können in Form von Paketen oder als Bytestrom ausgetauscht werden.

Ihre Benutzung ähnelt der von Netzwerk-Sockets, die Kommunikation ist aber schneller. Übertragene Daten müssen nicht den Umweg über den Netzwerk-Stack nehmen und geroutet werden. Sie werden stattdessen direkt vom Betriebssystemkern

[1] http://math.nist.gov/scimark2/
[2] Title 17 § 105 United States Code: http://www.copyright.gov/title17/92chap1.html#105
[3] http://en.wikipedia.org/w/index.php?title=Unix_domain_socket&oldid=385587800

übertragen, der dabei einige Kontextwechsel einspart, weil er zum Beispiel nicht in den Netzwerktreiber wechseln muss. Die Kommunikation kann blockierend oder nicht-blockierend mit Puffer stattfinden.

```
1  if ((s = socket(AF_UNIX, SOCKETTYPE, 0)) == -1) {
2     perror("socket");
3     exit(1);
4  }
5  local.sun_family = AF_UNIX;
6  strcpy(local.sun_path, SOCKPATH);
7  unlink(local.sun_path);
8
9  if (bind(s, (struct sockaddr *) &local, sockaunsize) == -1) {
10    perror("bind");
11    exit(1);
12 }
13
14 if (listen(s, _locCnt) == -1) {
15    perror("listen");
16    exit(1);
17 }
18
19 if ((remote = accept(s, 0, &sockaunsize)) == -1) {
20    perror("accept");
21    exit(1);
22 }
```

Listing 6.3: Beispielcode Unix Domain Sockets

Sockets werden über Inodes im Dateisystem adressiert. Listing 6.3 zeigt einen Ausschnitt aus einem C-Programm, das einen passiven Socket öffnet. In Zeile 1 wird mit einem Aufruf von `socket()` ein Socket geöffnet, der in Zeile 9 mit `bind()` an einen Dateinamen gebunden wird. Die dazu verwendete Datenstruktur `local` gibt außerdem den Typ des Sockets an: `AF_UNIX`. In Zeile 7 wurde mit `unbind()` sichergestellt, dass der in `SOCKPATH` gespeicherte Dateiname nicht belegt ist. Eine Datei mit diesem Namen würde gelöscht.

In Zeile 14 wird der Socket mit `listen()` als passiver Socket eingestellt, das heißt, dass der Socket Verbindungen annimmt. `accept()` in Zeile 19 wartet dann auf eine tatsächliche Verbindung. Andere Prozesse können Sockets an denselben Dateinamen binden, und sich dann mit dem im Beispiel erzeugten „lauschenden“ Socket verbinden.

Unix Domain Sockets wurden für die Linux-Implementierung der prioritätenbasierten Synchronisation genutzt, um eine Message-Passing-Funktionalität zwischen verschiedenen Locations zu simulieren.

6.4. Der PRET-Simulator

Für den in 4.8.2 beschriebenen PRET existiert ein in C++ geschriebener Simulator[4], der die zyklengenaue Simulation des Prozessors in einer Linux-Umgebung ermöglicht. Eine SystemC-Version[5] ist ebenfalls erhältlich. Er steht unter einer BSD-artigen Lizenz und kann frei verwendet werden.

Der Simulator simuliert sechs Hardware-Threads, für die je ein getrennt kompiliertes Programm im SPARC-ELF-Format benötigt wird. Unbenutzte Threads werden mit Dummy-Programmen belegt.

Mitgelieferte Scripte und Makefiles erleichtern die Kompilierung mit Hilfe des SPARC-Crosscompilers.

Sollen Programmcode und Daten zu Beginn der Ausführung in den 64kB umfassenden Scratchpad-RAM geladen werden, so kann dies durch Anlegen von Textdateien, die die zu ladenden Adressbereiche enthalten, veranlasst werden.

Der Kommunikation zwischen den Threads dient der 8MB große Shared Memory. Im C-Code werden dazu in allen Teil-Programmen volatile Zeiger auf fest vorgegebene Adressen definiert. Da in den 13 Zyklen, für die ein Thread während eines Umlaufs des Memory Wheels Zugriff auf den Hauptspeicher erlangt, immer nur ein Schreib- oder Lesezugriff möglich ist, ist die Shared-Memory-Kommunikation relativ langsam.

Installation und Verwendung des Simulators sind in Anhang B erklärt. Der im Paket enthaltene Debugger erlaubt die schrittweise Ausführung von Programmen sowie die Abarbeitung einer beliebigen Zahl von Zyklen. Die ausgeführten Befehle und aktuelle Registerinhalte können ausgegeben werden. Die Beobachtung ausgewählter einzelner Threads ist ebenfalls möglich. Der Debugger wurde für diese Arbeit nicht benötigt.

Die Hardware-Threads des PRET werden in den Abschnitten 6.6 und 6.7.1 zur besseren Unterscheidung von SC-Threads als *Locations* bezeichnet.

6.4.1. printf-Workaround

Bei Experimenten mit den beim PRET-Simulator mitgelieferten Beispielen stellte sich heraus, dass der Simulator die Ausgaben verschiedener Hardware-Threads miteinander vermengt. Während `printf()` normalerweise die Ausgaben puffert, bis ein Thread oder Prozess einen Zeilenumbruch ausgibt, schreibt der Simulator eine zufällige Anzahl Zeichen sofort auf den Bildschirm. Versuche, die Ausgaben der Teilprogramme in Dateien umzulenken, schlugen fehl. Vermutlich unterstützen die im Paket enthaltenen Bibliotheken dies nicht. Die korrekte Funktion von parallelisierten SC-Programmen kann jedoch ohne Ausgabe nicht überprüft werden.

Die Ausgaben in dedizierte Speicherbereiche zu schreiben und im Anschluss auszuwerten, ist nicht praktikabel, da so keine Deadlocks betrachtet werden können.

[4] `http://chess.eecs.berkeley.edu/pret/src/pret-1.0/pret_simulator.html`

[5] `http://www.systemc.org`

Außerdem müssten die Simulationen, damit es zur Ausgabe kommt, immer vollständig durchlaufen, was durchaus einige Minuten pro Simulation in Anspruch nehmen kann. Wird der Simulator aber beendet, so sind die Speicherinhalte verloren.

Daher wurde ein einfacher Workaround implementiert. Statt `printf()` verwenden die Programme `sprintf()`, um statt auf den Bildschirm in je einen Puffer im Shared Memory zu schreiben. Hat ein Programm seinen Puffer beschrieben, so setzt es ein Flag, das dies anzeigt. Hardware-Thread 5 arbeitet die Puffer umlaufend ab und löscht die Flags, um neue Schreibvorgänge zu gestatten, nachdem er den Pufferinhalt selbst mit `printf()` ausgegeben hat. Damit die Ausgaben der einzelnen Teilprogramme unterscheidbar werden, stellt er ihnen die Nummer des Hardware-Threads voran, von dem sie stammen.

```
1 #define printf(...) {while (*_printOutFlag==1){}; sprintf(_printOutPtr, __VA_ARGS__); *_printOutFlag=1;}
```

Listing 6.4: Überschreiben von `printf()`

Listing 6.4 zeigt die Präprozessordirektive in `sc-generic.h`, mit der alle Vorkommen von `printf()` im Code ersetzt werden. Sie muss *nach* der Einbindung von System-Header-Dateien geschehen, da Ersetzungen von `printf()` darin zu Fehlern beim Kompilieren führen können.

Wie man sieht, wird jedes Vorkommen von `printf()` durch Busy Waiting auf die Freigabe des Puffers, das anschließende `sprintf()` und das Setzen des Flags ersetzt. Dass die Anzahl der Parameter von `printf()` variabel ist, wird dem Präprozessor durch `(...)` mitgeteilt. Sie werden an Stelle von `__VA_ARGS__` eingefügt.

```
1  volatile char* _printOutPtr[5];
2  volatile char* _printOutFlag[5];
3
4  int main() {
5     int i;
6     for (i=0;i<5;i++) {
7        _printOutPtr[i] = (volatile char*) _printOutMemBase + (_printOutBufSize * i);
8        _printOutFlag[i] = (volatile char*) _printOutMemBase + _printOutBufSize -1 + (_printOutBufSize * i);
9        _printOutPtr[i][_printOutBufSize - 2] = '\0';
10       *_printOutFlag[i] = 0;
11    }
12    while(1) {
13       for (i=0;i<5;i++) {
14          if (*_printOutFlag[i] == 1) {
15             printf("%d: %s", i, _printOutPtr[i]);
16             *_printOutFlag[i] = 0;
17          }
18       }
19    }
20 }
```

Listing 6.5: Ausgabe aus Puffern

Listing 6.5 zeigt das Programm, das Hardware-Thread 5 ausführt. Die Adresse, bei der die Pufferstruktur beginnt, ist als `_printOutMembase` definiert. Die Zeiger sind als *volatile* qualifiziert, damit der Compiler keine Speicherzugriffe wegoptimiert, indem er gelesene Inhalte in einem Register aufbewahrt.

Dass ein Hardware-Thread für diesen Workaround belegt wird, ist prinzipiell akzeptabel. Schließlich müssen bei einer Simulation nicht alle sechs Threads genutzt werden. Die Übertragung sämtlicher Ausgaben über den Shared Memory verringert aber auch die Leistung des Simulators. DMA-Transfers, mit denen man die Ausgaben aus dem Scratchpad-RAM in den Hauptspeicher übertragen könnte, stehen noch nicht zur Verfügung [43].

Der Workaround erfordert auch kleinere Änderungen am SCmc-Code: Alle Ausgaben müssen mit einem Umbruch beendet werden. Aufeinander folgende Teilausgaben, wie in Listing 6.1, zu sehen sind nicht mehr möglich, da das Stringende im Puffer angezeigt sein muss. An vielen Stellen im Code wurden deshalb Umbrüche in die Ausgaben eingefügt. Kompliziertere Funktionen, zum Beispiel zur Ausgabe von Signalbelegungen mit den Namen der Signale, wurden in gepufferte Varianten umgeschrieben.

```
1  1: ==== TICK 2 STARTS, inputs = 00000, enabled = 033
2  0: Outputs OK.
3  1: ==== Inputs (id/name): <init>
4  0: ==== TICK 2 STARTS, inputs = A0000, enabled = 027
5  1: ==== Enabled (id/state):  0/_L_TICKEND,  1/_L146,  3/_L142,  4/_L152
6
7  0: ==== Inputs (id/name):  0/A
8  0: ==== Enabled (id/state):  0/_L_TICKEND,  1/_L244,  2/_L227,  4/_L251
9
10 1: AWAIT:    4/_L152
11 0: AWAIT:    4/_L251
12 1: determines R/2 certainly absent, waits
13 0: determines R/2 certainly absent, waits
```

Listing 6.6: Trace-Ausgabe bei Nutzung des printf-Workarounds

Abbildung 6.6 zeigt einen Ausschnitt aus einer Trace-Ausgabe eines auf dem PRET ausgeführten SCmc-Programms. Wie man sieht, ist die Ausgabe nicht synchronisiert, Zeile 2 stammt noch aus dem vorigen Tick. Einige Ausgaben sind durch die eingefügten Umbrüche unterbrochen, wie in Zeile 10 bis 13 zu sehen. Da Ausgaben derselben Location geordnet bleiben, bleiben Traces aber nachvollziehbar.

6.4.2. Deadline-Instruktion

Die in Abschnitt 4.8.2 bereits erwähnte Deadline-Instruktion erlaubt es, wie von Edwards et al. gefordert [41], das zeitliche Verhalten der CPU in Software zu kontrollieren. Sie fungiert als zyklengenauer Timer und ermöglicht es, der Ausführungszeit eines Abschnittes im Programmcode eine untere Schranke zu setzen. An ihr ange-

langt stoppt die Ausführung des Threads, bis die festgelegte Zahl an Prozessorzyklen verstrichen ist.

```
1  int main() {
2    DEAD(28);
3    volatile unsigned int * buf =
4        (unsigned int*)(0x3F800200);
5    unsigned int i = 0;
6    for (i = 0; ; i++ ) {
7      DEAD(26);
8      *buf = i;
9    }
10   return 0;
11 }
```

Listing 6.7: Producer

```
1  int main() {
2    DEAD(41);
3    volatile unsigned int * buf =
4        (unsigned int*)(0x3F800200);
5    unsigned int i = 0;
6    int arr[8];
7    for (i =0; i<8; i++)
8      arr[i] = 0;
9    for (i = 0; ; i++) {
10     DEAD(26);
11     register int tmp = *buf;
12     arr[i%8] = tmp;
13   }
14   return 0;
15 }
```

Listing 6.8: Consumer

Abbildung 6.2.: Anwendungsbeispiel Deadline-Instruktion (aus: Lickly et al. [43])

Die Listings 6.7 und 6.8 zeigen, wie mit Deadline-Instruktionen ein Producer und ein Consumer synchronisiert werden können. Sie bringen die durch das Memory Wheel konkurrenzfreien Speicherzugriffe in eine feste Reihenfolge.

Eine Variante der Deadline-Instruktion verwendet einen programmierbaren Phase-Locked-Loop-Timer zur Zeitmessung. Sie eignet sich zur Abstimmung des Systemverhaltens auf äußere Abläufe, wie zum Beispiel die Zeilenfrequenz eines Fernsehbilds [43], wird für die vorgestellten Implementierungen aber nicht benötigt.

Ist die WCET bekannt, so kann bei der Umsetzung eines Systems für eine reale Anwendung mittels Deadline-Instruktionen eine ticksynchrone, fest getaktete Ausführung ohne Barriers erreicht werden. Hier wurde jedoch zu Gunsten minimaler Ausführungszeit, und um die Ausführungsgeschwindigkeit bestimmen zu können, auf den Einsatz verzichtet.

Die in Abschnitt 5.3 beschriebenen Priority-Barriers einzusparen, wenn bekannt ist, wie lange ein Prioritätsschritt maximal dauert, brächte hingegen nur dann einen Vorteil, wenn die mit den Barriers eingesparte Rechenzeit größer ist als die durch unnötiges Warten verschwendete.

6.5. Code-Struktur

SC ist in mehrere C- und Header-Dateien aufgeteilt: `sc.h` und `sc-generic.h` enthalten Deklarationen und Definitionen, wobei fast alle Makros und alle Operatoren in letztgenannter Datei definiert sind. `sc.c` beinhaltet allgemeine Funktionen und `sc-main.c` die `main`-Funktion mit Initialisierung, Lesen der Ein- und Ausgaben und

Prüfung derselben. Aus der `main`-Funktion in `sc-main.c` wird die Tick-Funktion `tick()` aufgerufen, also das eigentliche SC-Programm.

Für SCmc wurde an dieser Aufteilung nichts geändert. Lediglich Signaldefinitionen und Einstellungen wurden in zusätzliche Header-Dateien ausgelagert.

6.6. Signalbasierte Synchronisation auf dem PRET

Für die signalbasierte Synchronisation, beschrieben in Kapitel 5.2, ist der Shared Memory von Vorteil. Signale und Certainty-Matrix können dort abgelegt werden.

6.6.1. Signale und Certainty-Matrix

SC verwaltet Signale platzsparend in Bitvektoren. Einen Bitvektor zu verändern erfordert aber einen Lese- und einen anschließenden Schreibzugriff, daher müssten Mechanismen zum gegenseitigen Ausschluss implementiert werden, um diese platzsparende Speicherung in SCmc zu erhalten.

Assembler-Instruktionen zur atomaren Bitmanipulation gibt im SPARC-Befehlssatz [1] nicht. Zwar existieren Anweisungen zur Vertauschung einer Speicherstelle mit einem Registerinhalt oder ähnlichem. Ob diese auf dem PRET verwendet werden können, ist aber unklar, zumal sie ein bestimmtes Speichermodell, *Total Store Ordering*, voraussetzen, während der PRET mit dem Memory Wheel eigene Wege geht. Ohnehin dienen solche Instruktionen eher der Realisierung von Locking-Mechanismen, was, wie im letzten Kapitel erläutert, vermieden werden soll.

In SCmc wurden die Bitvektoren daher aufgegeben und die Signale in Arrays abgelegt. Um der Speicherausrichtung der SPARC-Architekturen gerecht zu werden und mehrfache Speicherzugriffe zu vermeiden, wurden für Daten im Shared Memory nur 32Bit-Typen wie `long` und `int` verwendet. Der Speicherbedarf für Signale ist in SCmc also gegenüber SC verzweiunddreißigfacht.

Operatoren, die auf Signal-Arrays zugreifen, wie `PRESENT`, `EMIT` oder `AWAIT` wurden indirekt durch die Anpassung von Makros wie `_sigAdd()` auf Arrays umgestellt. Während das Setzen und Prüfen von Signalen dadurch sogar etwas vereinfacht wurde, mussten Operationen auf Sets von Signalen neu geschrieben werden. Sie wurden als normale Funktionen implementiert. Geschwindigkeitseinbußen durch Funktionsaufrufe nicht zu erwarten, da der GCC selbstständig Funktionen inline einbinden kann, wenn dies Vorteile bringt.

Die Funktionsweise der Certainty-Matrix wurde in Abschnitt 5.2.1 bereits erklärt. Die Matrix ist ein Array im Shared Memory. Listing 6.9 zeigt in den Zeilen 1 und 4 die Deklaration der Zeiger auf die Matrix und den lokalen Teil derselben. `_setPartType` ist bereits ein Array von `unsigned long`s. Die komplizierte Deklaration ist notwendig, um das Array trotz der Angabe einer konkreten Speicheradresse komfortabel zweidimensional indizierbar zu machen.

In `_setMembase` ist eine feste Speicheradresse, die allen Locations bekannt ist, gespeichert. Der nicht volatil deklarierte Array-Zeiger `localCertaintyMatrix` zeigt

```
1  volatile _setPartType (*const certaintyMatrix)[__sigCnt] =
2     (volatile _setPartType (*const)[__sigCnt]) (_setMemBase + (sizeof(sigset)));
3
4  _setPartType (*const localCertaintyMatrix)[__sigCnt] =
5     (_setPartType (*const)[__sigCnt]) (_setMemBase + (sizeof(sigset)))
6     + ((__idCnt-1) * _SCMC_LOCATION);
7
8  int _localCMStartIdx = ((__idCnt-1) * _SCMC_LOCATION);
9  int _localCMEndIdx =    ((__idCnt-1) * (_SCMC_LOCATION + 1));
10
11 int _certaintyRow[__idCnt-1];
```

Listing 6.9: Deklaration der Certainty-Matrix

auf einen Speicherbereich, der von der Nummer der Location, definiert als Makro `_SCMC_LOCATION`, abhängt. Dieser Speicherbereich beinhaltet den lokalen Teil der Matrix. Die lokalen Abschnitte sämtlicher Locations bilden zusammen das Array, auf das `certaintyMatrix` zeigt. Dieser Zeiger muss volatil deklariert sein. Beide Zeiger sind außerdem konstant deklariert, um Fehler zu verhindern.

Das in Zeile 11 definierte Array `_certaintyRow` dient der in Abschnitt 5.2.1 erwähnten Indirektion beim Schreibzugriff. Es speichert zu jeder Priorität den Index der durch den jeweiligen Prozess verwendeten Zeile. Beim Wechsel der Priorität mit `PRIO` müssen die Einträge der neuen und der alten Priorität in diesem Array vertauscht werden.

Die Indizes, die in den Zeilen 8 und 9 definiert werden, dienen der Leistungssteigerung in Funktionen, die die Certainty-Matrix durchlaufen. Sie überspringen den lokalen Teil der Matrix, wie anhand der Funktion `_sigCertaintyCheck()` in Listing 6.10 beispielhaft gezeigt.

```
1  int _sigCertaintyCheck(int s) {
2     int i;
3     int end = _locCnt * (_idCnt -1);
4     _procCertainAll(s);
5
6     for (i=0; i < _localCMStartIdx; i++) {
7        if (!certaintyMatrix[i][s]) {
8           return 0;
9        }
10    }
11    for (i=_localCMEndIdx; i < end; i++) {
12       if (!certaintyMatrix[i][s]) {
13          return 0;
14       }
15    }
16    return 1;
17 }
```

Listing 6.10: `_sigCertaintyCheck()` (gekürzt)

`_sigCertaintyCheck()` prüft, ob ein Signal CERTAINLY ABSENT ist. In Zeile 4 wird mit dem Aufruf `_procCertainAll(s)`, wie in Abschnitt 5.2.3 erklärt, das geprüfte Signal in der lokalen Matrix als „sicher“ eingetragen.

Zur Verwaltung der Einträge in der Certainty-Matrix wurden diverse weitere Funktionen erstellt, die es zum Beispiel erlauben, die Einträge eines Threads oder eines Sets von Threads auf „sicher“ zu stellen oder die Matrix zur Fehlersuche komplett auszugeben. Solche Debug-Ausgaben enthalten in der Regel auch Speicheradressen, so dass die Indizierung überprüft werden kann.

6.6.2. Barriers

```
1  volatile unsigned int* const tickStartBarrier =
2        ((volatile unsigned int* const) _barrierMemBase)
3        + _SCMC_LOCATION_MAX + 1;
4
5
6  void _hitBarrier(volatile unsigned int* const barrier) {
7     int i, flag;
8     barrier[_locNum] = 1;
9     if (_locNum != 0) {
10       while (barrier[_locNum] == 1) {}
11    }
12
13    else {
14       while(1) {
15          flag = 1;
16          for (i=1; i < _locCnt; i++) {
17             if (barrier[i] == 0) flag = 0;
18          }
19          if (flag) {
20             for (i=0; i < _locCnt; i++) {
21                barrier[i] = 0;
22             }
23             return;
24          }
25       }
26    }
27 }
```

Listing 6.11: `_hitBarrier()` (gekürzt)

Barriers werden in allen Implementierungen zur Synchronisation der Ticks eingesetzt. Vor und nach dem Aufruf der Tickfunktion müssen sich alle Locations an einer Barrier treffen.

Die Barriers sind wie die Certainty-Matrix Arrays im Shared Memory. Sie enthalten ein Element pro Location. Der Anfang von Listing 6.11 zeigt, wie ein volatiler konstanter Zeiger auf ein solches Array an der als als `_barrierMemBase` definierten Adresse angelegt wird.

Die Funktion `_hitBarrier()` setzt den Inhalt des zur eigenen Location gehörenden Elements in Zeile 8 auf `1`. Außer auf Location 0 wird dann in Zeile 10 durch aktiv gewartet, bis dieser `0` wird. Location 0 „bewacht“ die Barrier und wartet darauf, dass

alle Locations an der Barrier angekommen sind. Anschließend setzt sie die Einträge alle auf `0`.

Dadurch dass jede Location außer 0 ihr eigenes Array-Element zur Verfügung hat und auch nur dieses beobachtet, werden Race-Conditions bei mehrfacher Verwendung derselben Barrier hintereinander vermieden.

Die Überwindung der Barrier per Busy Waiting hat auf dem PRET keine Nachteile, da das Memory-Wheel zwischen den Zugriffen, die 13 Zyklen dauern, eine Wartezeit von 65 Zyklen erzwingt [43]. Eine Location kann also in 78 Zyklen nur einmal eine Speicherstelle überprüfen und dabei keine anderen Locations blockieren.

Um Energie zu sparen wäre es auf einem anderem System allerdings sinnvoll, Energiesparfunktionen des Prozessors zu nutzen und entsprechende Wartezeiten in die Schleifen einzubauen.

```
1         Start Simulation
2 ----------------------------------------------------
3 1: #### RUN 0 STARTS ################################
4 0: #### RUN 0 STARTS ################################
5 1: RESET:    0/(null)
6 0: RESET:    0/(null)
7 1: reset automaton
8 0: reset automaton
9 1: ==== TICK 0 STARTS, inputs = 00000
10 0: ==== TICK 0 STARTS, inputs = 00000
11 1: ==== Inputs (id/name): <init>
12 0: ==== Inputs (id/name): <init>
13 1: Location 1 stops at barrier 3ffe0008.
14 0: Location 0 guards barrier 3ffe0008.
15 0: Location 0 opens gate at barrier 3ffe0008.
16 1: Location 1 crosses barrier 3ffe0008.
17 0: FORK:     4/_L_INIT
18 1: FORK:     4/_L_INIT
```

Listing 6.12: Trace mit Barrier-Debug-Ausgaben

Für den Trace in Abbildung 6.12 wurden Debug-Ausgaben in `_hitBarrier()` eingeschaltet. In den Zeilen 13 bis 16 geben die Locations ihre Ankunft und Überwindung der Barriers inklusive der Speicheradresse des Barrier-Arrays aus.

6.6.3. Blockierende Threads

Aufgrund der in Abschnitt 5.2.2 beschriebenen Probleme wurde diese Methode nicht implementiert.

Abgesehen von den theoretischen Problemen würden blockierende Threads in SC aber auch ganz konkrete Umsetzungsprobleme mit sich bringen. Das Makro `PRESENT`, gezeigt in Listing 6.13, wäre zum Beispiel nicht implementierbar, da es zur Verwendung als Bedingung in einem `if`-Statement gedacht ist. Aus Ausdrücken in runden Klammern kann man jedoch nicht mit `goto` heraus springen. Da das selbe für Funktionen gilt, brächte es auch nichts, `PRESENT` in eine Funktion auszulagern.

```
#define PRESENT(s)   \
   (trace3tc("PRESENT:", "determines %s/%d %s\n",  \
   s2signame[s], s, _sigContains(signals, s) ? "present" : "absent") \
   (_checkPRESENTc(s) _sigContains(signals, s)))
```

Listing 6.13: Das Makro PRESENT

6.6.4. Blockierende Locations

Die signalbasierte Synchronisation mit blockierenden Locations, beschrieben in Abschnitt 5.2.3, wurde in einer einfachen Form, ohne LCR-Algorithmus oder Initialisierungsvektoren, implementiert. Die Certainty-Matrix muss mit a priori bekannten Werten initialisiert werden, wozu eine Funktion `getCertainty()`, die vor jedem Tick aufgerufen wird, verwendet wird. Die Werte werden als zweidimensionales Array im Code jeder Location abgelegt.

```
1: ==== TICK 3 STARTS, inputs = A0000, enabled = 033
1: ==== Inputs (id/name):  0/A
0: ==== TICK 3 STARTS, inputs = 0B000, enabled = 023
1: ==== Enabled (id/state):  0/_L_TICKEND,  1/_L146,  3/_L142,  4/_L152

0: ==== Inputs (id/name):  1/B
0: ==== Enabled (id/state):  0/_L_TICKEND,  1/_L244,  4/_L251

1: _sigCertaintyCheck[2/R] returns 1
0: _sigCertaintyCheck[2/R] returns 1
1: AWAIT:    4/_L152
0: AWAIT:    4/_L251
1: determines R/2 certainly absent, waits
0: determines R/2 certainly absent, waits
1: AWAIT:    3/_L142
1: determines B/1 present, proceeds
0: _sigCertaintyCheck[4/X] finds 0 in row 4, returns 0 (certaintyMatrix[4][4]=3fff0874)
1: JOIN:     1/_L146
0: PRESENT: determines X/4 uncertain, blocks
1: joins
1: EMIT:     1/_L146
0: _sigCertaintyCheck[4/X] finds 0 in row 4, returns 0 (certaintyMatrix[4][4]=3fff0874)
1: emits X/4
0: _sigCertaintyCheck[4/X] finds 0 in row 4, returns 0 (certaintyMatrix[4][4]=3fff0874)
0: PRESENT: determines X/4 present
0: JOIN:     1/_L244
0: joins
0: RET:      1/_L244
0: returns
0: EMIT:     1/_L244
0: emits O/3
1: ==== TICK 3 terminates after 4 instructions.
0: ==== TICK 3 terminates after 4 instructions.
```

Listing 6.14: Trace mit Certainty-Debug-Ausgaben

In Listing 6.14 ist ein Trace eines einzelnen Ticks mit aktivierten Debug-Ausgaben in der Funktion `_sigCertaintyCheck()` gezeigt. Sie gibt `1` zurück, wenn das geprüfte

Signal CERTAINLY ABSENT ist, andernfalls `0`. Auch hier wird die Speicheradresse zur Kontrolle mit ausgegeben.

```
1 #define _blockOnSignal(s, functionname)
                        \
2    if ( !(_sigContains(signals, s) || _certAbsent(s)) ) {
                    \
3       trace2t(functionname, "determines %s/%d uncertain, blocks\n", s2signame[s], s); \
4       while ( !(_sigContains(signals, s) || _certAbsent(s)) ) {
                    \
5       } \
6    }
```

Listing 6.15: `_blockOnSignal()`

Listing 6.15 zeigt die Definition der Funktion `_blockOnSignal()`, die das Verhalten eines SCmc-Programms mit blockierenden Locations bestimmt. Sie prüft in Zeile 2 zunächst, ob das Signal PRESENT oder CERTAINLY ABSENT ist. Letzteres geschieht durch einen Aufruf von `_certAbsent()`, der ein Alias für die in Listing 6.10 gezeigte Funktion `_sigCertaintyCheck()` ist. Ist beides nicht der Fall, so ist das Signal „unsicher“, und die Funktion wiederholt die Prüfung in Zeile 4, bis das Signal PRESENT oder CERTAINLY ABSENT ist.

Ein Aufruf von `_blockOnSignal()` am Anfang von `PRESENTELSE`, `PRESENTEMIT`, `AWAIT` und `AWAITI` sorgt dafür, dass diese Operatoren erst entscheiden, wenn der Zustand eines Signals sicher bekannt ist, beziehungsweise, dass bei Prüfung eines unsicheren Signals blockiert wird. Da `_blockOnSignal()` erst terminiert, wenn die Prüfung erfolgreich beendet wurde, bleibt die Location bei „unsicheren“ Signalen blockiert.

Außerdem sorgt in `PAUSE`, `TERM` und Operatoren, die ein `PAUSE` implizit enthalten, ein Aufruf von `_sigCertainAll()` dafür, dass der entsprechende Thread seine Zeile in der Certainty-Matrix auf „sicher“ stellt.

Angepasst wurde auch die Prüfung auf Kausalitätsfehler. Denn nur Signale, die CERTAINLY ABSENT sind, können auch als getestet gelten.

Mit den zusätzlichen Operatoren `WONTEMIT(s)` und `WONTEMITANY` kann ein Thread frühzeitig anzeigen, dass er ein bestimmtes Signal nicht oder gar kein Signal mehr emittieren wird. Ein weiterer neuer Operator `TIDYUP` trägt in jedem Tick `1` für alle inaktiven Threads ein, wie in Abschnitt 5.2.3 beschrieben. Aufgrund der Tatsache, dass er nie terminiert und mit einer Priorität niedriger als der aller Threads mit einem `FORKE`, aber höher als der aller anderen Threads ausgeführt werden muss, ist er aber von geringem Nutzen.

Das Makro `PRESENT` wird in SC in eine Zeile von kommagetrennten Ausdrücken expandiert. In C werden solche nacheinander ausgewertet, aber nur das Ergebnis des letzten, ganz rechts stehenden, verwendet. SC macht sich dies zu Nutze, gibt Trace-Meldungen aus und trägt das geprüfte Signal für die Kausalitätsprüfung ein, bevor der letzte Ausdruck den Status des Signals liefert. In SCmc muss `PRESENT` jedoch blockieren können. Die dazu benötigte `while`-Schleife kann in einem Klammerausdruck

```
1  int _present(int s) {
2     if ( !(signalsPtr[s] || _sigCertaintyCheck(s)) ) {
3        while ( !(signalsPtr[s] || _sigCertaintyCheck(s)) ) {}
4     }
5     if (signalsPtr[s]) {
6        return 1;
7     }
8     else {
9        return 0;
10    }
11 }
```

Listing 6.16: `_present()` (gekürzt)

```
1 #define PRESENT(s) (_present(s) ? 1 : \
2    (instrCntIncrc _checkSIG_ID(s, _SC_SIG_MAX), _sigAdd(_presence_tested, s), 0))
```

Listing 6.17: Blockierenden Variante des Makros `PRESENT`

nicht verwendet werden. Daher wurde die Definition von `PRESENT` angepasst und durch den Aufruf der Funktion `_present()` in `sc.c` ergänzt. Bei zukünftigen Änderungen muss dieser Funktion besondere Aufmerksamkeit geschenkt werden. Sie enthält Code aus anderen Makros in `sc-generic.h`, der zum Teil angepasst wurde. Das Makro `PRESENT` und die Funktion `_present()` sind in Listing 6.16 und Listing 6.17 gezeigt.

6.7. Prioritätenbasierte Synchronisation

Während die signalbasierte Synchronisation in den Signalabfragen ansetzt, findet die prioritätenbasierte Synchronisation im Dispatcher statt. Der Dispatcher wurde zwischen der Auswahl der nächsten auszuführenden Priorität mit `selectCid()` und dem tatsächlichen `goto`-Sprung in den entsprechenden Thread um den Aufruf einer neuen Funktion `_waitForPrio()` erweitert. Der erweiterte Dispatcher ist in Listing 6.18 gezeigt.

```
1 #define dispatch()   selectCid(); _waitForPrio(_cid); _goto(_deref(_pc[_cid]))
```

Listing 6.18: Prioritäten synchronisierender Dispatcher

`_waitForPrio()` nimmt als Parameter die nächste lokal auszuführende Priorität an und ruft `_hitPrioBarrier()` auf. Dort findet die in Abschnitt 5.3 beschriebene Abstimmung über die Prioritäten zwischen den Locations statt. Location 0 „bewacht“ die Priority-Barrier ebenso wie normale Barriers und wählt unter den angegeben Prioritäten die höchste aus. Freie Prioritäten werden also übersprungen. Locations,

die auf eine niedrigere Priorität warten, laufen sofort nach Überwindung wieder in die Priority-Barrier hinein, weil `_waitForPrio()` `_hitPrioBarrier()` so lange aufruft, bis die lokale nächste Priorität der globalen entspricht.

In den nächsten beiden Abschnitten wird die Implementierung prioritätenbasierter Synchronisation auf dem PRET und unter Linux beschrieben.

6.7.1. PRET

```
1  int _hitPrioBarrier(volatile idtype* const prioBarrier,
2        volatile unsigned int* const barrier,
3        idtype minPrio) {
4     idtype currentPrio = minPrio;
5     int i, flag;
6
7     if (_locNum != 0) {
8        prioBarrier[_locNum] = minPrio;
9        barrier[_locNum] = 1;
10       while (barrier[_locNum] == 1) {}
11       currentPrio = prioBarrier[0];
12    }
13
14    else {
15       while(1) {
16          flag = 1;
17          for (i=1; i < _locCnt; i++) {
18             if (barrier[i] == 0) flag = 0;
19          }
20          if (flag) {
21             for (i=1; i < _locCnt; i++) {
22                if (prioBarrier[i] > currentPrio) currentPrio = prioBarrier[i];
23             }
24             prioBarrier[0] = currentPrio;
25             for (i=0; i < _locCnt; i++) {
26                barrier[i] = 0;
27             }
28             return currentPrio;
29          }
30       }
31    }
32    return currentPrio;
33 }
```

Listing 6.19: `_hitPrioBarrier()` für Shared-Memory-Kommunikation (gekürzt)

Die Shared-Memory-Implementierung der Priority-Barrier-Funktion ist in Listing 6.19 gezeigt. Sie entspricht einer normalen Barrier, die wie in Abschnitt 6.6.2 beschrieben um ein Array im Shared Memory erweitert wurde. In dieses tragen die Locations ihre nächste lokal auszuführende Priorität ein, bevor sie anzeigen, dass sie die Barrier erreicht haben. Stellt Location 0 in Zeile 17 und 18 fest, dass alle anderen die Barrier erreicht haben, wählt sie in Zeile 21 und 22 die höchste aus und trägt diese im Array ein. Die Öffnung der Barrier in der Schleife ab Zeile 25 veranlasst die anderen Locations, diese zu lesen und zurückzuliefern.

```
1  0: AWAIT:    3/WaitA
2  1: Location 1 stops at priority barrier 3ffe0000, suggested priority=0.
3  0: initial pause
4  0: Location 0 guards priority barrier 3ffe0000, suggested priority=1.
5  0: Location 0 opens gate at priority barrier 3ffe0000, priority=1->1.
6  1: Location 1 crosses priority barrier 3ffe0000, new priority=1.
7  0: CALL:     1/ABMain
8  1: Location 1 stops at priority barrier 3ffe0000, suggested priority=0.
9  0: calls ABRJOIN
10 0: PAUSE:    1/ABMain
11 0: pauses, active = 03
12 0: Location 0 guards priority barrier 3ffe0000, suggested priority=0.
13 0: Location 0 opens gate at priority barrier 3ffe0000, priority=0->0.
14 1: Location 1 crosses priority barrier 3ffe0000, new priority=0.
15 0: Location 0 guards barrier 3ffe0008.
16 1: Location 1 stops at barrier 3ffe0008.
17 0: Location 0 opens gate at barrier 3ffe0008.
18 1: Location 1 crosses barrier 3ffe0008.
19 0: ==== TICK 0 terminates after 9 instructions.
20 1: ==== TICK 0 terminates after 7 instructions.
```

Listing 6.20: Trace mit Priority-Barrier-Debug-Ausgaben (PRET)

Listing 6.20 zeigt einen Ausschnitt aus einem Trace mit aktivierten Debug-Ausgaben in `_hitPrioBarrier()` und `_hitBarrier()`. In Zeile 2 meldet Location 1, dass sie die Priorität 0 vorschlägt, also den Tick beenden möchte. Dies wird, an den Meldungen in Zeile 4 bis 6 erkennbar, verwehrt. dass Location 1 läuft daher erneut in die Barrier, zu sehen an der Meldung in Zeile 8. Die Ausgaben in den Zeilen 12 bis 14 zeigen, dass schließlich die Priorität 0 ausgewählt wird. Beide Locations synchronisieren sich im Anschluss über die normale Barrier zum Ende des Ticks.

Auch die Implementierung prioritätenbasierter Synchronisation auf dem PRET verwaltet Signale im Shared Memory. Darüber hinausgehende Änderungen an den Operator-Makros wie in Abschnitt 6.6 mussten aber nicht vorgenommen werden. Weil durch den in Abschnitt 6.4.1 beschriebenen Workaround `printf()` gegen einen Ausdruck mit Semikolon ersetzt wird, wurden die Ausgaben in `PRESENT` jedoch deaktiviert.

6.7.2. Linux

Die Linux-Implementierung von SCmc wurde als universellere Umgebung entwickelt, mit der sich Algorithmen für Message-Passing über FIFO-Kanäle testen lassen, bevor die Umsetzung auf konkreter Hardware versucht wird. Die Implementierung erlaubt zwar keine präzise Simulation, ist aber sehr viel schneller, als der PRET-Simulator.

Die Locations werden hier durch Linux-Systemprozesse simuliert. Die leichtgewichtigeren POSIX-Threads kommen nicht in Frage, da sie vollständig im Kontext eines gemeinsamen Systemprozesses ausgeführt werden. Um den Locations eigene Speicherbereiche, zum Beispiel für die Prozesstabellen, zuzuweisen, wäre eine aufwändige Umstrukturierung des gesamten SCmc-Codes notwendig gewesen.

Die Location-Prozesse sind als separate Programme ausgeführt und werden in einzeln ausführbare Dateien kompiliert. Gegenüber der Abspaltung von Prozessen mittels der Systemfunktion `fork()` aus einem zentralen Programm bietet dies mehrere Vorteile: Zum einen ist sichergestellt, dass jede Location wirklich nur ihren eigenen Speicherinhalt nutzt. Variablen, die vor der Ausführung von `fork()` initialisiert wurden, wären sonst auf allen Child-Prozessen vorab mit einem Wert belegt. Zum anderen lässt sich die Ausgabe getrennter Programme leichter umlenken, so dass jede Location getrennt beobachtet werden kann. Der in Abschnitt 6.1 erwähnte Profiler `gprof` schreibt zudem immer in die selbe Ausgabedatei, wenn man die Programme nicht von verschiedenen Arbeitsverzeichnissen aus startet. Darüber hinaus werden durch getrennte C-Dateien Probleme mit uneindeutigen Labels verhindert.

Zur Kommunikation untereinander nutzen die Prozesse die in Abschnitt 6.3 beschriebenen Unix Domain Sockets. Die Einrichtung der Sockets erfolgt in der `main`-Funktion in `sc-main.c`. Jede Location hat einen Socket, über den sie mit Location 0 kommuniziert, Location 0 verwaltet hingegen mehrere Sockets in einem Array. Die Verbindungen sind also sternförmig. Die Ausführung beginnt erst, wenn alle Verbindungen zustande gekommen sind. Bei Beendigung des Programms, egal ob ordnungsgemäß oder durch Fehler, bricht die Ausführung aller Locations durch Socket-Fehler ab.

```
1  #define _SCMC_BARR_ARV "A"          // Arrival at barrier
2  #define _SCMC_BARR_OPN "O"          // Barrier open
3  #define _SCMC_BARR_PRI "P"          // Arrival at prio barrier
4  #define _SCMC_BARR_PRO "Q"          // Prio barrier open
5
6  #define _SCMC_MSG_SIG "S"              // Signal present message
7  #define _SCMC_MSG_DEL "D"              // Signal deleted message
8  #define _SCMC_MSG_VAL "V"              // Valued Signal message
```

Listing 6.21: Definitionen von Nachrichtentypen

Übertragen werden ausschließlich Nachrichten mit festgelegter Maximallänge. Sie bestehen aus einem Buchstaben, der einen der in Listing 6.21 gezeigten Nachrichtentypen angibt, und gegebenenfalls einer Priorität oder Signalnummer. Die Sende- und Empfangsoperationen mit Fehlererkennung wurden in eigenen Funktionen gekapselt. Als Beispiel ist in Listing 6.22 die Funktion `_sendSig()` gezeigt, die eine Signalnachricht an Location 0 beziehungsweise von dort an alle anderen Locations sendet. Signalnachrichten werden grundsätzlich über Location 0 verteilt.

Mit Message-Passing fällt die Implementierung von Barriers leicht: Jede Location, die eine Barrier erreicht, sendet einfach eine entsprechende Nachricht an Location 0 und wartet dann blockierend auf die Freigabe der Barrier durch eine Antwort. Location 0 wartet blockierend auf Nachrichten von allen anderen Locations, bevor sie eine Freigabenachricht sendet.

Das Warten auf die Ankunft eines Signals findet nicht nur unter Linux ressourcenschonend statt, in dem der Prozess „schläft“, bis eine Nachricht für ihn eintrifft.

```
1  void _sendSig(int sig) {
2     char str[_SCMC_MSGLEN+1];
3     int j;
4     sprintf(str, "%s%d", _SCMC_MSG_SIG, sig);
5     if (_locNum != 0) {
6        _send(str, s);
7     }
8     else {
9        for (j=1; j < _locCnt; j++) {
10          _send(str, sockets[j]);
11       }
12    }
13 }
```

Listing 6.22: `_sendSig()` (gekürzt)

Der in Abschnitt 4.6.3 beschriebene XMOS-Prozessor kann zum Beispiel Hardware-Threads warten lassen, bis eine Nachricht eintritt, und verbraucht dabei weniger Strom.

Allerdings muss auch der Austausch von Signalen über Nachrichten stattfinden. Dies erfordert einen asynchronen Mechanismus, entweder durch nebenläufigen Empfang, oder durch Puffer. Für die Linux-Implementierung wurde angenommen, dass auch auf dem simulierten System Puffer in ausreichender Größe zur Verfügung stehen.

Die in Puffern gespeicherten Signalnachrichten werden zwischen den Prioritätsschritten empfangen. Die Funktion `_hitPrioBarrier()` gestaltet sich daher deutlich komplizierter als die Shared-Memory-Variante.

Listing 6.23 enthält den Kopf und den Teil der Funktion, der für alle Locations außer 0 ausgeführt wird. In Zeile 11 und 12 wird die Nachricht mit der lokalen Priorität gesendet, danach blockiert die Location in Zeile 15 bis zum Empfang einer Nachricht. Ist die empfangene Nachricht eine Barrier-Freigabe (Zeile 17ff), so wird darin eine Priorität erwartet und zurückgegeben. Handelt es sich um eine Signalnachricht, so wird in Zeile 26 bis 31 die Signalnummer ausgewertet und das Signal lokal als PRESENT eingetragen.

Listing 6.24 zeigt den Code für Location 0. In Zeile 4 bis 37 wird umlaufend auf Nachrichten von allen Locations gewartet. Wurde eine Nachricht empfangen, die anzeigt, dass eine Location die Barrier erreicht hat, so wird in Zeile 27 bis 36 die enthaltene Priorität ausgewertet und von derselben Location nichts mehr empfangen. Handelt es sich um eine Signalnachricht, so wird in Zeile 17 die Signalnummer gelesen und die Nachricht in Zeile 20 bis 24 an alle Locations bis auf die sendende weitergeleitet. Die Prüfung in Zeile 18 verhindert, dass Signale, die schon als PRESENT bekannt sind, mehrfach gesendet werden. Die benötigte Puffergröße wird dadurch beschränkt. Die Prüfung findet ebenfalls in `EMIT` statt.

Wurde in Zeile 38 festgestellt, dass alle Locations die Barrier erreicht haben, so wird diese in der Schleife ab Zeile 40 freigegeben. Die Freigabe-Nachricht enthält die

```
1  int _hitPrioBarrier(idtype minPrio) {
2     idtype currentPrio = minPrio;
3     int i, j, flag;
4     int sigid;
5     char str[_SCMC_MSGLEN+1];
6     char arrived[_locCnt];
7     memset(&str, 0, sizeof(str));
8     memset(&arrived, 0, sizeof(arrived));
9
10    if (_locNum != 0) {
11       sprintf(str, "%s%d", _SCMC_BARR_PRI, minPrio);
12       _send(str, s);
13
14       while(1) {
15          _blockingReceive(str, s);
16
17          if (strncmp(_SCMC_BARR_PRO, str, strlen(_SCMC_BARR_PRO)) == 0) {
18             if (strlen(str) <= strlen(_SCMC_BARR_PRO)) {
19                printf("Error: Location %d expected priority, received plain %s instead.\n", _locNum, str);
20                exit(1);
21             }
22             currentPrio = atoi(&str[strlen(_SCMC_BARR_PRO)]);
23             return currentPrio;
24          }
25          else if (strncmp(_SCMC_MSG_SIG, str, strlen(_SCMC_MSG_SIG)) == 0) {
26             if (strlen(str) <= strlen(_SCMC_MSG_SIG)) {
27                printf("Error: Location %d received signal message %s, w/o signal id.\n", _locNum, str);
28                exit(1);
29             }
30             sigid = atoi(&str[strlen(_SCMC_MSG_SIG)]);
31             signals[sigid] = 1;
32          }
33       }
34    }
```

Listing 6.23: `_hitPrioBarrier()` für Message-Passing - Location n (gekürzt)

nächste auszuführende Priorität. Locations, die auf eine niedrigere Priorität warten, laufen also auch hier direkt wieder in die Barrier hinein.

Die Annahme, dass die Barrier freigegeben werden kann und keine Signalnachrichten mehr verarbeitet werden müssen, wenn alle Locations sie erreicht haben, ist korrekt. Denn die Kommunikation über Unix Domain Sockets erhält die FIFO-Ordnung der Nachrichten. Daher wird eine Nachricht, die das Erreichen der Barrier anzeigt, immer zuletzt empfangen. Die Signalnachrichten werden sofort in `EMIT` abgesendet, also niemals zwischen dem Erreichen der Barrier und der Freigabe derselben.

Abbildung 6.3 zeigt die Ausschnitte aus den Trace-Ausgaben eines Programms mit zwei Locations.

Da mit der Linux-Implementierung nur Locations simuliert werden sollen, enthält sie lediglich rudimentäre Fehlerbehandlung. Ein Prozess, der einen Fehler bemerkt, terminiert einfach. Eine „Poisonous-Pill“-Nachricht, mit der ein Prozess anderen mitteilt, dass sie sich beenden sollen, existiert ebenfalls nicht, daher kann es nach Ablauf der Simulation harmlose Fehlermeldungen durch Abbruch der Socketverbindungen geben.

```
1    else {
2       while(1) {
3          flag = 1;
4          for (i=1; i < _locCnt; i++) {
5             if (arrived[i] == 1) {
6                continue;
7             }
8             else {
9                flag = 0;
10            }
11            _blockingReceive(str, sockets[i]);
12            if (strncmp(_SCMC_MSG_SIG, str, strlen(_SCMC_MSG_SIG)) == 0) {
13               if (strlen(str) <= strlen(_SCMC_MSG_SIG)) {
14                  printf("Error: Location %d received signal message %s, w/o signal id.\n", _locNum, str);
15                  exit(1);
16               }
17               sigid = atoi(&str[strlen(_SCMC_MSG_SIG)]);
18               if (signals[sigid] == 0) {
19                  signals[sigid] = 1;
20                  for (j=1; j < _locCnt; j++) {
21                     if (j != i) {
22                        _send(str, sockets[j]);
23                     }
24                  }
25               }
26            }
27            else if (strncmp(_SCMC_BARR_PRI, str, strlen(_SCMC_BARR_PRI)) == 0) {
28               if (strlen(str) <= strlen(_SCMC_BARR_PRI)) {
29                  printf("Error: Location %d expected priority, received plain %s from %d instead.\n",
                         _locNum, str, i);
30                  exit(1);
31               }
32               if (atoi(&str[strlen(_SCMC_BARR_PRI)]) > currentPrio) {
33                  currentPrio = atoi(&str[strlen(_SCMC_BARR_PRI)]);
34               }
35               arrived[i] = 1;
36            }
37         } // for
38         if (flag) {
39           sprintf(str, "%s%d", _SCMC_BARR_PRO, currentPrio);
40            for (i=1; i < _locCnt; i++) {
41               _send(str, sockets[i]);
42            }
43            return currentPrio;
44         }
45      } // while(1)
46   } // Location 0
47 }
```

Listing 6.24: `_hitPrioBarrier()` für Message-Passing - Location 0 (gekürzt)

Überlegungen zur Realisierbarkeit

Sind auf dem realen System keine Puffer vorhanden, so könnte zumindest ähnlich wie in Abschnitt 6.4.1 eine Location allein zur Kommunikation verwendet werden und permanent Nachrichten empfangen und puffern.

Location 0 kann, wenn es weder Reinkarnation noch valued Signals gibt, während eines Prioritätsschrittes von jeder Location jedes Signal einmal empfangen. Außerdem kann jede Location mitteilen, dass sie die Barrier erreicht hat. Ihr Puffer muss

```
1  ==== L0: TICK 3 STARTS, inputs = 00R00, enabled = 021
2  ==== Inputs (id/name):  2/R
3  ==== L0: Enabled (id/state):  0/_L_TICKEND,  4/_L212
4  Location 0 guards priority barrier, suggested priority=4.
5  Location 0 opens gate at priority barrier, priority=4->4.
6  AWAIT:    4/_L212 determines R/2 present, proceeds
7  ABORT:    4/_L212 disables 012, enabled = 021
8  FORK:     4/_L212 forks 1/AB, active = 023
9  FORKE:    4/_L212 continues at ABOMain
10 Location 0 guards priority barrier, suggested priority=4.
11 Location 0 opens gate at priority barrier, priority=4->4.
12 AWAIT:    4/ABOMain initial pause
13 Location 0 guards priority barrier, suggested priority=1.
14 Location 0 opens gate at priority barrier, priority=1->2.
15 Location 0 guards priority barrier, suggested priority=1.
16 Location 0 opens gate at priority barrier, priority=1->3.
17 Location 0 guards priority barrier, suggested priority=1.
18 Location 0 opens gate at priority barrier, priority=1->2.
19 Location 0 guards priority barrier, suggested priority=1.
20 Location 0 opens gate at priority barrier, priority=1->1.
21 FORK:     1/AB forks 3/WaitA, active = 013
22 FORKE:    1/AB continues at ABMain
23 Location 0 guards priority barrier, suggested priority=3.
24 Location 0 opens gate at priority barrier, priority=3->3.
25 AWAIT:    3/WaitA initial pause
26 Location 0 guards priority barrier, suggested priority=1.
27 Location 0 opens gate at priority barrier, priority=1->1.
28 CALL:     1/ABMain calls ABRJOIN
29 JOIN:     1/ABMain waits
30 Location 0 guards priority barrier, suggested priority=0.
31 Location 0 opens gate at priority barrier, priority=0->0.
32 Location 0 guards barrier.
33 Location 0 opens the gate.
```

Listing 6.25: Location 0

```
==== L1: TICK 3 STARTS, inputs = 00R00, enabled = 021
==== L1: Enabled (id/state):  0/_L_TICKEND,  4/_L101
Location 1 stops at priority barrier, suggested priority=4.
Location 1 crosses priority barrier, priority=4->4.
AWAIT:    4/_L101 determines R/2 present, proceeds
ABORT:    4/_L101 disables 014, enabled = 021
FORK:     4/_L101 forks 2/AB, active = 025
FORKE:    4/_L101 continues at ABOMain
Location 1 stops at priority barrier, suggested priority=4.
Location 1 crosses priority barrier, priority=4->4.
AWAIT:    4/ABOMain initial pause
Location 1 stops at priority barrier, suggested priority=2.
Location 1 crosses priority barrier, priority=2->2.
FORK:     2/AB forks 3/WaitB, active = 015
FORKE:    2/AB continues at ABMain
Location 1 stops at priority barrier, suggested priority=3.
Location 1 crosses priority barrier, priority=3->3.
AWAIT:    3/WaitB initial pause
Location 1 stops at priority barrier, suggested priority=2.
Location 1 crosses priority barrier, priority=2->2.
JOIN:     2/ABMain waits
Location 1 stops at priority barrier, suggested priority=0.
Location 1 crosses priority barrier, priority=0->1.
Location 1 stops at priority barrier, suggested priority=0.
Location 1 crosses priority barrier, priority=0->3.
Location 1 stops at priority barrier, suggested priority=0.
Location 1 crosses priority barrier, priority=0->1.
Location 1 stops at priority barrier, suggested priority=0.
Location 1 crosses priority barrier, priority=0->0.
Location 1 stops at barrier.
Location 1 crosses barrier.
Location 1 stops at barrier.
Location 1 crosses barrier.
```

Listing 6.26: Location 1

Abbildung 6.3.: Traces mit Priority-Barrier-Debug-Ausgaben (Linux)

also $(S+1)\cdot(L-1)$ Nachrichten fassen können, wobei S die Anzahl der Signale und L die Anzahl der Locations ist. Jede andere Location muss, da sämtliche Kommunikation über Location 0 abgewickelt wird, maximal eine Nachricht pro Signal puffern können, was einen Puffer für bis zu S Nachrichten voraussetzt. Freigabenachrichten zu puffern ist nicht notwendig.

Gepufferte FIFO-Kommunikation kann auch auf Shared-Memory-Systemen implementiert werden. Ein einfacher Ringpuffer, dessen Pufferfüllstand an der Differenz von Schreib- und Lesezeiger abgelesen wird, wurde für den PRET-Simulator implementiert, um die Machbarkeit zu demonstrieren. Der gegenseitige Ausschluss wird dadurch garantiert, dass nur eine Location den Schreibzeiger versetzen darf, während die andere den Lesezeiger versetzt.

6.8. Nicht implementierte Funktionalität

Die Switch-Case-Logik von SC wurde mit SCmc nicht getestet. Keine der Implementierungen beherrscht valued Signals. Prüfungen auf Signale des vorigen Ticks funktionieren nicht in der Implementierung blockierender Locations aus Abschnitt 6.6.4, hierzu müsste `PRESENTPRE` analog zu `PRESENT` umgeschrieben werden. Eine zusätzliche Barrier, die die Signale erst speichert, wenn alle Locations sie erreicht haben, wird in `setPre` bereits beachtet.

PRESENT und PRESENTPRE liefern bei der Implementierung prioritätenbasierter Synchronisation auf dem PRET keine Trace-Ausgaben. PRESENTELSE, PRESENTPREELSE, PRESENEMIT, PAUSEG, JOINE, PPAUSE, JPPAUSE, und SUSTAIN wurden mit keiner Implementierung getestet, allerdings sind hier keine größeren Probleme zu erwarten. Suspension und Reincarnation wurden in SCmc nicht implementiert. Lokale Signale werden in SC praktisch als globale behandelt. Wurden Threads, die ein lokales Signal gemeinsam nutzen, auf verschiedene Locations verteilt, wird Reincarnation nicht funktionieren.

Während FORKs und JOINs wie in Abschnitt 5.4 beschrieben leicht zwischen den Locations zu synchronisieren sind, sind die Operatoren ISAT und ISATCALL in SCmc nicht location-übergreifend umsetzbar, da sie Zugriff auf entfernte Thread-Tabellen bräuchten.

Funktionen zum Verschieben eines Threads zwischen den Locations sind nicht vorgesehen und würden das Verhalten des Gesamtsystems schwer vorhersehbar machen. Funktionen zum Erzeugen eines Threads auf einer anderen Location und zum Warten auf dessen Terminierung wurden nicht implementiert. Beides kann mit Hilfe vorhandener SCmc-Operatoren und Signalen erreicht werden.

Die Simulation des PRET bleibt bei Erkennung eines Kausalitätsfehlers stehen, anstatt abzubrechen, da die Funktion `exit()` noch nicht durch das Assemblerkommando zum Abbruch ersetzt wurde.

6.9. Vergleich der Implementierungen

Bei allen Implementierungen wurde versucht, die Komplexität durch zusätzlichen Code möglichst wenig zu erhöhen und keine Indeterminismen einzubauen, die sich nicht aus dem Konzept selbst ergeben. Konstrukte zum gegenseitigen Ausschluss wie Semaphore wurden bewusst nicht eingesetzt, um die gegenseitige Beeinflussung der Locations gering zu halten. Auf dynamische Datenstrukturen konnte bei allen Implementierungen verzichtet werden, die Linux-Implementierung setzt allerdings Nachrichtenpuffer voraus.

Zur Umsetzung der signalbasierten Synchronisation mussten zahlreiche Funktionen angepasst werden. Die Certainty-Matrix muss aufwändig adressiert und durchlaufen werden und benötigt zudem viel Speicherplatz. Durch das Memory Wheel des PRET entstehen Wartezeiten beim Speicherzugriff, die sich stärker auswirken, je größer die Matrix ist. Während einer einfachen Signalprüfung müssen, wenn das Signal nicht PRESENT ist, sämtliche Einträge einer Spalte in der Matrix gelesen oder geschrieben werden. Die Länge einer Spalte ergibt sich aus dem Produkt der maximalen PIDs auf einer Location und der Zahl der Locations. Pausiert oder terminiert ein Thread, werden so viele Einträge geschrieben, wie Signale existieren. Die Implementierung skaliert also verhältnismäßig schlecht.

Für PRE-Funktionen sind weitere Änderungen und eine zusätzliche Barrier notwendig.

Die Implementierung prioritätenbasierter Synchronisation mit Hilfe von Priority-Barriers ist allgemein deutlich einfacher als die signalbasierte, und somit auch weniger fehlerträchtig. Die meisten Makros können hier unverändert bleiben, der größte Aufwand liegt in der Programmierung der Barriers. Probleme mit Strong und Weak Abortions oder `JOIN` gibt es nicht. Die Größe der Barriers in Shared Memory hängt lediglich von der Zahl der Locations ab. Zur Überwindung einer Priority-Barrier greift eine Location mindestens viermal auf den Speicher zu. Durch Busy Waiting sind weitere Zugriffe möglich, doch verzögern diese die Ausführung nur an der Barrier.

Je nach Architektur ließe sich Busy Waiting an Priority-Barriers durch energiesparendes Warten auf eine Nachricht ersetzen oder mit einem Hardware-Synchronizer implementieren. Für die komplexe Certainty-Matrix gibt es keine Hardware-Unterstützung.

Die Linux-Implementierung erforderte einen gewissen zusätzlichen Aufwand zur Umsetzung der Socket-Kommunikation. Davon abgesehen ist sie die einfachste, da keine Komplikationen mit Ausgaben auftreten. Strenggenommen wäre es nicht einmal nötig gewesen, die Speicherung der Signale in Vektoren aufzugeben. Die Änderung wurde lediglich aus der PRET-Implementierung übernommen, um den Code leichter vergleichen zu können.

Die Umsetzungen für den PRET beruhen auf der Annahme, dass der Speicherzugriff fair ist. Ist das nicht der Fall, muss ohne Busy Waiting implementiert werden. Die Linux-Implementierung lässt sich direkt auf Prozessoren mit Message-Passing-Mechanismen umsetzen, bei ungepufferter Kommunikation muss ein Kern als Kommunikationskanal programmiert werden.

Durch Priority-Barriers synchronisierte Ausführungen sind, abgesehen vom Abarbeiten der Nachrichtenschlangen bei Message Passing, nicht vom Status oder der Zahl der Signale abhängig und besser vorhersehbar als signalsynchronisierte.

6.10. Mögliche Optimierungen

Für Shared-Memory-Implementierungen sollten Speicherzugriffe so weit wie möglich reduziert werden. Wenn ein Signal PRESENT ist, könnte daher auf Einträge in der Certainty-Matrix verzichtet werden.

Bei einer Signalprüfung müssten die Spalten der Matrix nicht immer wieder bis zur ersten `0` lesend durchlaufen werden. Hier würde es genügen, nur den entsprechenden Eintrag Feld in der Matrix wiederholt zu prüfen, wenn auch im Wechsel mit dem Eintrag im Signalarray, denn das Signal könnte während der Prüfung PRESENT werden. Durch lokale Kopien der Signalarrays und der Certainty-Matrix könnten weitere Lesezugriffe eingespart werden.

Bei Message-Passing-Kommunikation könnten durch selektive Zustellung Nachrichten eingespart werden. Signale müssen nur an die Locations gesendet werden, die diese auch verarbeiten. Caspi et al. [15] beschreiben eine einfache Analyse. Ein Parser könnte diese durchführen und Vorkommen von `EMIT` durch rein lokales Setzen

des Signals oder zielgerichtetes Senden ersetzen. Sinnvoll wäre es auch, Nachrichten nicht zur Laufzeit mit Stringoperationen auszuwerten, sondern zur Compilezeit fest zu kodieren.

Eine mögliche Weiterentwicklung von SCmc wäre die Einführung von Kontrollsignalen, die die Synchronisation von `FORK` und `JOIN` transparent erlauben.

7. Experimentelle Ergebnisse

In dieser Arbeit wurden qualitativ unterschiedliche Synchronisationsmethoden implementiert, die sich nur schwer quantitativ vergleichen lassen. Auch zwischen den beiden Implementierungen prioritätenbasierter Synchronisation für den PRET und für Linux bestehen große Unterschiede. Letztere verwendet Message Passing, und das mit gegenüber dem schnellen SC-Code hohem Overhead.

Mit dem zyklengenauen PRET-Simulator ist es allerdings möglich, die Gesamtausführungszeit eines Programms genau zu bestimmen. Die Angabe der ausgeführten Instruktionen lässt zudem Rückschlüsse auf die Zahl der Speicherzugriffe zu. Listing 7.1 zeigt die Ausführungsstatistik eines ABRO-Testlaufs mit signalbasierter Synchronisation. Der `printf`-Workaround und der Scratchpad-RAM waren aktiviert.

```
1  --------------------------------------------------
2           Simulation Statistics
3  --------------------------------------------------
4  Cycles executed : -1
5
6  thread[0], # cycles: 50112138
7  thread[0], # instructions: 1583021
8  thread[1], # cycles: 50112138
9  thread[1], # instructions: 1715517
10 thread[2], # cycles: 50112132
11 thread[2], # instructions: 8336809
12 thread[3], # cycles: 50112132
13 thread[3], # instructions: 8336806
14 thread[4], # cycles: 50112132
15 thread[4], # instructions: 8336803
16 thread[5], # cycles: 50112132
17 thread[5], # instructions: 1074401
```

Listing 7.1: Ausführungsstatistik PRET-Simulator

Aufgrund der gleichmäßigen Rechenzeitverteilung durchlaufen alle Threads die gleiche Zahl an Zyklen. Wie viele Instruktionen ein Thread in dieser Zeit ausführen kann, hängt aber primär von der Zahl der Speicherzugriffe ab. Da ein Zugriff bis zu 90 Zyklen dauern kann, ist jede andere Instruktion um ein Vielfaches schneller.

Die Threads 2 bis 4 wurden nicht benutzt und mit einem Dummy-Programm geladen, das nichts tut. Offenbar ist es in 50112132 Zyklen möglich, 8336803 No Operations (NOPs) auszuführen. Das ergibt 6 Zyklen pro NOP-Instruktion, was zur umlaufenden Ausführung von 6 Threads passt.

Thread 5 hingegen, der permanent aus dem Speicher liest, während er das Programm aus Listing 6.5 ausführt, konnte nur 1074401 Instruktionen ausführen. Die

Threads 0 und 1 haben mehr als eineinhalb mal so viele geschafft, was darauf schließen lässt, dass sie zwar häufig auf den Speicher zugreifen, aber nicht so intensiv wie Thread 5.

Um genauer zu untersuchen, wie sich Ausgaben, Speicherzugriffe und die Nutzung des Scratchpad-RAMs (SPM) auswirken, wurde zunächst das SC beiliegende nicht verteilte ABRO unter verschiedenen Bedingungen für 5 Ticks in 2 Durchgängen auf dem PRET-Simulator ausgeführt. Tabelle 7.1 zeigt die Ergebnisse bei aktivierter Trace-Ausgabe. Die erste Spalte enthält die Zahl der Zyklen, die die Simulation benötigte. Die folgenden beiden Spalten, beschriftet mit „Ins. 0“ und „Ins 1“ geben die Zahl der Instruktionen für Thread 0 und Thread 1 an. „Z/I“ steht für den gerundeten Quotienten aus Zyklen und Instruktionen. Die rechten beiden Spalten enthalten die ebenfalls gerundeten Ergebnisse des Kommandos `time`, das die Ausführungszeit des Simulators gemessen hat. Zur Ermittlung der Werte in der zweiten Zeile wurden sowohl Daten als auch Instruktionen in den Scratchpad-RAM geladen.

	PRET					`time`	
	Zyklen	Ins. 0	Ins. 1	Z/I 0	Z/I 1	usr	sys
ohne SPM	71165742	624261	11845747	114	6	0m7.9s	1m28.8s
mit SPM	28020996	624261	4654956	45	6	0m38.1s	0.016s

Tabelle 7.1.: Scratchpad-Messergebnisse ABRO mit Ausgabe

Aus beiden Ausführungen ist ersichtlich, dass Thread 1 lediglich NOP-Instruktionen ausgeführt hat. Die Ausführung ohne Scratchpad-RAM war erwartungsgemäß langsamer, sie hat zweieinhalb mal so lange gedauert, wie die mit aktiviertem Scratchpad-RAM. Ohne letzteren müssen auch alle Instruktionen aus dem Shared Memory geholt werden, was die Zyklen pro Instruktion von 45 auf 114 erhöht. Besonders auffallend sind aber die hohen Werte für „sys“, die verbrauchte Systemzeit. Möglicherweise nutzt der Simulator Kernel-Funktionen auf eine sehr ineffiziente Art. Bei der Ausführung ohne Scratchpad-RAM war die eigentliche Rechenzeit im Userspace, abzulesen in der Spalte „usr“, sogar geringer als bei der Ausführung mit Scratchpad. Ohne genauere Kenntnis des Simulators kann dies nicht erklärt werden.

	PRET					`time`	
	Zyklen	Ins. 0	Ins. 1	Z/I 0	Z/I 1	usr	sys
ohne SPM	4521342	39661	738347	114	6	0m6.25s	0m0.02s
I. im SPM	2323800	39661	372090	59	6	0m3.25s	0m0.03s
I. & D. im SPM	2009616	39661	319726	51	6	0m2.77s	0m0.02s

Tabelle 7.2.: Scratchpad-Messergebnisse ABRO ohne Ausgabe

Tabelle 7.2 zeigt Ergebnisse für Ausführungen des selben Programms ohne SPM, mit Instruktionen im SPM und mit Instruktionen und Daten im SPM. Die Ergebnisse für Ausführungen ohne Instruktionen aber mit Daten im SPM unterscheiden

sich nicht messbar von solchen ganz ohne Nutzung des SPMs. Beim Kompilieren wurden die Makros `_SC_SUPPRESS_ERROR_DETECT` und `_SC_NOTRACE` gesetzt, um die Ausgabe durch das SC-Programm zu unterdrücken.

Wie man sieht, fanden die Ausführungen erheblich schneller statt. Kernel-Zeit wurde praktisch keine verbraucht, was den Verdacht, dass es Probleme mit der Ausgabe des Simulators gibt, erhärtet. Aus den Werten geht klar hervor, dass die Nutzung des SPMs für Instruktionen die mittlere Ausführungszeit von Instruktionen etwa halbiert. Werden zusätzlich lokale Daten im SPM abgelegt, beschleunigt das nochmal um etwa ein Achtel.

Offenbar ist es also sinnvoll, Vergleichsmessungen mit abgeschalteter Ausgabe und unter Nutzung des Scratchpad-RAMs durchzuführen.

	PRET					`time`	
	Zyklen	Ins. 0	Ins. 1	Z/I 0	Z/I 1	usr	sys
CM	4312608	37830	37830	114	114	0m5.85s	0m0.02s
CM (SPM)	2066610	37915	48797	55	42	0m3.25s	0m0.03s
PB	4409736	38682	38682	114	114	0m6.32s	0m0.03s
PB (SPM)	2094198	40050	48545	52	43	0m3.19s	0m0.04s

Tabelle 7.3.: Vergleichsmessungen mit verteiltem ABRO

Um signalbasierte und prioritätenbasierte Synchronisierung vergleichen zu können, wurde ein verteiltes ABRO mit erweitertem `JOIN`-Konstrukt, zu sehen in Listing 7.2 und 7.3, für je 5 Ticks in 2 Durchgängen auf dem Simulator ausgeführt. Sämtliche Ausgaben wurden deaktiviert, so dass der `printf`-Workaround keinen Einfluss auf die Ergebnisse haben kann.

Tabelle 7.3 zeigt das Resultat. „CM“ steht für Certainty-Matrix, als signalbasierte Synchronisation, „PB“ für Priority-Barrier, also prioritätenbasierte Synchronisation. Aus den Messwerten geht hervor, dass die Programme nahezu gleichschnell ausgeführt werden. Um eindeutige Unterschiede ausmachen zu können, wären umfangreichere Programme notwendig, die deutlich mehr Signale verarbeiten.

Alle PRET-Programme wurden mit dem `sparc-elf-gcc` und den Parametern `-O3 -msoft-float -save-temps -lm` kompiliert. Der PRET-Simulator wurde in einer Linux-Umgebung auf einem Mobilprozessor Intel CoreDuo T2500 (Yonah) bei einer festen Taktfrequenz von 2,00 GHz ausgeführt.

Programme, die direkt auf einem x86-Desktop-Prozessor ausgeführt werden, sind im Gegensatz zu auf dem PRET-Simulator ausgeführten extrem schnell. Für brauchbare Vergleichsmessungen müssen sie daher wesentlich mehr Ticks durchlaufen. Dies wurde durch eine einfache Erweiterung der `main`-Funktion um eine Schleife zur Wiederholung des gesamten Programms erreicht. So modifiziert kann die Linux-Implementierung von SCmc mit SC verglichen werden. Bei SCmc wurde allerdings die zeitaufwändige Socket-Initialisierung von der Wiederholung ausgenommen. Alle Programme wurden wie oben mit `_SC_SUPPRESS_ERROR_DETECT` und `_SC_NOTRACE` kompiliert, um die Ausgabe zu unterdrücken.

```
1   TICKSTART(4);
2
3   do {
4       FORK(AB, 1);
5       FORKE(ABOMain);
6
7   AB:
8       FORK(WaitA, 3);
9       FORKE(ABMain);
10
11  WaitA:
12      AWAIT(A);
13      TERM;
14
15  ABMain:
16      CALL(ABRJOIN);
17      EMIT(O);
18      TERM;
19
20   ABRJOIN:
21      while (ABchildren != 0) {
22         if (PRESENT(X)) ABchildren--;
23         else PAUSE;
24      }
25      JOIN;
26      RET;
27
28  ABOMain:
29      AWAIT(R);
30      ABORT;
31
32  } while (1);
33
34  TICKEND;
```

Listing 7.2: Verteiltes ABRO – Thread0

```
1   TICKSTART(4);
2
3   do {
4     FORK(AB, 1);
5     FORKE(ABOMain);
6
7   AB:
8     FORK(WaitB, 3);
9     FORKE(ABMain);
10
11  WaitB:
12    AWAIT(B);
13    TERM;
14
15  ABMain:
16    JOIN;
17    EMIT(X);
18    TERM;
19
20  ABOMain:
21    AWAIT(R);
22    ABORT;
23
24  } while (1);
25
26  TICKEND;
```

Listing 7.3: Verteiltes ABRO – Thread1

Tabelle 7.4 zeigt die mit `time` gemessenen Rechenzeiten im Userspace und im Kernel. Als zentrales Programm wurde das in SC enthaltene ABRO unverändert verwendet und in jeder Wiederholung wie oben in 2 Durchläufen zu je 5 Ticks ausgeführt. In der Tabelle sieht man, dass seine Ausführung praktisch vollständig im Userspace stattfindet und die verbrauchte Rechenzeit zur Zahl der Wiederholungen proportional ist.

	zentral		verteilt	
	usr	sys	usr	sys
1000	0m0.006s	0m0.001s	0m0.131s	0m0.531s
10000	0m0.070s	0m0.002s	0m1.205s	0m5.242s
100000	0m0.717s	0m0.002s	0m10.084s	0m43.522s

Tabelle 7.4.: ABRO auf x86

Als verteiltes Programm wurde das bekannte verteilte ABRO aus Listing 7.2 und 7.3 verwendet. Auch seine Ausführungszeit erscheint in etwa proportional zur

Zahl der Wiederholungen. Die Werte sind vermutlich durch die einmalige Socket-Initialisierung, die von `time` mitgemessen wird, minimal verzerrt. Auffällig ist, dass die beanspruchte Kernel-Zeit ebenfalls mit der Zahl der Wiederholungen steigt. Das war zu erwarten, denn die Socket-Kommunikation findet über den Kernel statt. Die verteilte Ausführung ist etwa sechzig Mal langsamer als die zentrale.

Der enorme Overhead durch Socket-Kommunikation und vermutlich auch Parsen der Nachrichten macht es schwer, festzustellen, ab wann sich eine Verteilung lohnt. Auf einem realen System würden Nachrichten von der Hardware übertragen und möglicherweise hartkodiert. Die Linux-Implementierung kann daher nur als Umgebung zur Evaluation von Algorithmen eingesetzt werden, aber nur sehr bedingt für Leistungsvergleiche.

Um in SCmc den Rechenaufwand in Zuständen künstlich erhöhen zu können, wurden, wie in Abschnitt 6.2 beschrieben, Funktionen, die Fourier-Transformationen berechnen, in die Zustände einiger Abwandlungen von ABRO eingefügt. Ausreichend „verteuert“ wird ein Programm natürlich schneller ausgeführt, wenn es auf mehrere Systemprozesse, und damit Prozessorkerne, verteilt wird. Es wäre aber auch möglich, durch schrittweise Anpassung des künstlichen Rechenaufwands zu ermitteln, ab welchem Verhältnis zwischen Rechenaufwand im Programm und Aufwand für Messagepassing sich eine Verteilung lohnt. Im Rahmen dieser Arbeit wurden hierzu jedoch keine weiteren Untersuchungen angestellt.

Alle x86-Programme wurden mit `gcc -O3 -save-temps -lm` kompiliert und auf dem schon genannten Mobilprozessor ausgeführt.

8. Zusammenfassung und Ausblick

Synchronous C erlaubt als makrobasierte C-Erweiterung leichtgewichtige und deterministische Nebenläufigkeit. Die grafische Modellierungssprache SyncCharts kann direkt in SC übersetzt werden. Bei der Übersetzung wird die Struktur eines SyncCharts weitgehend erhalten, nebenläufige Konstrukte jedoch sequentialisiert. Mit dieser Arbeit wurden Kommunikations- und Synchronisationsmethoden vorgestellt und implementiert, die die parallele Ausführung synchroner Programme ermöglichen. SC wird durch sie zu SCmc.

Andere Ansätze wurden betrachtet und festgestellt, dass diese überwiegend von über asynchrone Links verteilten Systemen ausgehen. Hier ist hingegen die Ausführung auf einem Multicore-Prozessor beabsichtigt. Aufgrund der schnellen und zuverlässigen Verbindung der Kerne untereinander treten Probleme, die von verteilten Systemen bekannt sind, nicht auf.

Da die genauen Möglichkeiten von den vorhandenen Mechanismen abhängen, wurden verschiedene Multicore-Prozessoren verglichen. Dabei wurde deterministischem Verhalten besondere Aufmerksamkeit gewidmet. Denn während synchrone Sprachen gut zur Programmierung reaktiver Systeme geeignet sind, führen viele Prozessoren bei der Ausführung neue Unvorhersehbarkeiten ein. Die Prozessoren haben sich einander überschneidende Eigenschaften, die eine Einordnung erschweren. Andererseits stechen innovative Konzepte heraus. Interessante Plattformen konnten identifiziert und eine für diese Arbeit geeignete ausgewählt werden.

Für den PRET, einen auf der SPARC-Architektur basierenden Prozessor mit vorhersagbarem zeitlichen Verhalten und einer besonderen, Fairness und gegenseitigen Ausschluss garantierenden Speicherarchitektur, existiert ein zyklengenauer Simulator. Dieser wurde genutzt, um auf Shared-Memory-Kommunikation basierend signalbasierte und prioritätenbasierte Synchronisation zu implementieren und zu vergleichen. Probleme mit der Ausgabe des Simulators wurden durch einen Workaround umgangen, so dass der Programmablauf trotzdem verfolgt werden kann.

Eine Linux-Implementierung prioritätenbasierter Synchronisation ermöglicht zudem die Evaluation von Algorithmen für Plattformen, die Message-Passing bieten. Eingebaute Wiederholschleifen und vorbereitete Shellscripte ermöglichen komfortable Laufzeitmessungen. Der Overhead durch die eingesetzte Unix-Socket-Kommunikation kann allerdings Ergebnisse verzerren.

Obwohl noch keine aussagekräftigen Experimente mit den implementierten Methoden durchgeführt werden konnten, ist bereits absehbar, dass die signalbasierte zu Problemen führt. Eine tiefgehende Betrachtung von Initialisierungsproblemen,

der Möglichkeit zum Deadlock, eventuell unvorhersehbarem zeitlichen Verhalten und Einschränkungen bezüglich der Übersetzbarkeit von SyncCharts lassen die signalbasierte Synchronisation wenig attraktiv erscheinen. Eine Ausprägung, das Blockieren von einzelnen Threads, konnte bereits früh als untauglich erkannt und verworfen werden.

Die signalbasierte Synchronisation nutzt das in Prioritäten gespeicherte Wissen über kausale Abhängigkeiten im Programm. Sie ist einfach zu implementieren, eignet sich gut für Message Passing und erhält zudem ein Höchstmaß an Determinismus.

Über die Betrachtung der Synchronisationsmethoden hinaus wurden erste konzeptionelle Ansätze zur Modifikation des existierenden SyncCharts-Compilers erläutert. Spätere Arbeiten könnten diese implementieren, so dass aus SyncCharts unter Ausnutzung ihrer inhärenten Nebenläufigkeit verteilter SC-Code generiert wird.

Als mögliche Optimierung wäre trotz der engen Kopplung eine Verringerung des Kommunikationsaufkommens sinnvoll. Die Synchronisation von Hierarchie-Operatoren wie `FORK` und `JOIN` zwischen verschiedenen Kernen könnte durch automatisches Einfügen von in SC-Code geschriebenen Konstrukten erleichtert werden. Auch die Verwendung von Valued Signals zu diesem Zweck oder die Einführung interner Steuersignale wäre denkbar. Letzteres würde die Unterschiede zwischen zentral ausgeführten SC- und verteilen SCmc-Programmen weiter verringern.

Da die Synchronisation, die ein zerlegtes synchrones Programm erfordert, immer einen zusätzlichen Aufwand bedeutet, sollte untersucht werden, wann sich die Verteilung lohnt. Zukünftige Arbeiten könnten den Versuch unternehmen, SCmc auf anderen Plattformen zu realisieren. Der XMOS XS1-G4 kann zum Beispiel Threads energiesparend auf Ereignisse oder Nachrichten warten lassen. Sein internes Kommunikationsnetz eröffnet zudem die Möglichkeit, sehr effizientes Message-Passing zu betreiben und so vielleicht schon bei geringen parallelisierbaren Anteilen eine beschleunigte Ausführung zu erreichen. Durch den Vergleich verschiedener Plattformen und Implementierungen könnte untersucht werden, welche auf dem Prozessor vorhandenen Mechanismen sich am besten zur Unterstützung der synchronen Programmierung eines Multicore-Prozessors eignen.

Literaturverzeichnis

[1] *The SPARC Architecture Manual Version 8.* SPARC International Inc., 535 Middlefield Road, Suite 210, Menlo Park, CA 94025, USA, 1992. – `http://www.sparc.org/standards/V8.pdf`

[2] Amdahl, Gene M.: Validity of the single processor approach to achieving large scale computing capabilities. In: *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, Spring Joint Computer Conference.* New York, NY, USA : ACM, 1967, S. 483–485. – `http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf`

[3] Amende, Torsten: *Synthese von SC-Code aus SyncCharts*, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Diplomarbeit, Mai 2010. – `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/tam-dt.pdf`

[4] Andalam, Sidharta ; Roop, Partha ; Girault, Alain ; Traulsen, Claus: PRET-C: A New Language for Programming Precision Timed Architectures. In: *Proocedings of the Workshop on Reconciling Performace with Predictability (RePP), Embedded Systems Week.* Grenoble, France, Oktober 2009

[5] André, Charles: Semantics of SyncCharts / I3S Laboratory. Sophia-Antipolis, France, April 2003 (ISRN I3S/RR–2003–24–FR). – Forschungsbericht

[6] Aubry, Pascal ; Guernic, Paul L.: On the Desynchronization of Synchronous Applications. In: *ICSE'96: Proceedings of the 11th International Conference on Systems Engineering.* Las Vegas, Nevada, USA, Juli 1996

[7] Awerbuch, Baruch: Complexity of Network Synchronization. In: *J. ACM* 32 (1985), Nr. 4, S. 804–823. – ISSN 0004-5411

[8] Beeck, Michael von d.: A Comparison of Statecharts Variants. In: Langmaack, H. (Hrsg.) ; Roever, W. P. de (Hrsg.) ; Vytopil, J. (Hrsg.): *Formal Techniques in Real-Time and Fault-Tolerant Systems* Bd. 863, Springer-Verlag, 1994, S. 128–148

[9] Benveniste, Albert ; Caillaud, Benoît ; Guernic, Paul L.: Compositionality in Dataflow Synchronous Languages: Specification and Distributed Code Generation. In: *Information and Computation* 163 (2000), Nr. 1, S. 125 – 171. – ISSN 0890-5401

[10] Berry, Gérard: *The Esterel v5 Language Primer, Version v5_91.* Centre de Mathématiques Appliquées Ecole des Mines and INRIA, 06565 Sophia-Antipolis: , 2000. – ftp://ftp-sop.inria.fr/esterel/pub/papers/primer.pdf

[11] Berry, Gérard ; Cosserat, Laurent: The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In: *Seminar on Concurrency, Carnegie-Mellon University* Bd. 197, Springer-Verlag, 1984, S. 389–448. – ISBN 3-540-15670-4

[12] Berry, Gérard ; Gonthier, Georges: The Esterel Synchronous Programming Language: Design, Semantics, Implementation. In: *Science of Computer Programming* 19 (1992), Nr. 2, S. 87–152

[13] Berry, Gérard ; Sentovich, Ellen M.: An Implementation of Constructive Synchronous Programs in POLIS. In: *Formal Methods in System Design* 17 (2000), Nr. 2, S. 135–161. – ISSN 0925-9856

[14] Burns, Alan ; Wellings, Andy: *Real-Time Systems and Programming Languages.* Addison Wesley, 2001. – 3. Auflage. – ISBN 0201729881

[15] Caspi, Paul ; Girault, Alain ; Pilaud, Daniel: Automatic Distribution of Reactive Systems for Asynchronous Networks of Processors. In: *IEEE Transactions on Software Engineering* 25 (1999), Nr. 3

[16] Chow, C.M.Edmund ; S.Y.Tong, Joyce ; Dayaratne, M.W.Sajeewa ; Roop, Partha S. ; Salcic, Zoran: RePIC - A New Processor Architecture Supporting Direct Esterel Execution / University of Auckland. 2004. – School of Engineering Report No. 612

[17] Closse, Etienne ; Poize, Michel ; Pulou, Jacques ; Venier, Patrick ; Weil, Daniel: Saxo-rt: Interpreting Esterel Semantic on a Sequential Execution Structure. In: *Electronic Notes in Theoretical Computer Science* 65 (2002), Nr. 5, S. 80–94. – SLAP'2002, Synchronous Languages, Applications, and Programming (Satellite Event of ETAPS 2002). – ISSN 1571-0661

[18] Courtois, P. J. ; Heymans, F. ; Parnas, D. L.: Concurrent Control with "Readers" and "Writers". In: *Commun. ACM* 14 (1971), Nr. 10, S. 667–668. – ISSN 0001-0782

[19] Craven, Stephen ; Patterson, Cameron ; Athanas, Peter: Configurable Soft Processor Arrays Using the OpenFire Processor. In: *International Conference on Military and Aerospace Programmable Logic Devices*, 2005, S. 7–9. – http://www.ccm.ece.vt.edu/papers/craven_2005_MAPLD05_openfire.pdf

[20] Dayaratne, M. W. S. ; Roop, Partha S. ; Salcic, Zoran: Direct Execution of Esterel Using Reactive Microprocessors. In: *Proceedings of Synchronous Languages, Applications, and Programming (SLAP'05)*, April 2005

[21] Edwards, Stephen: Concurrency and Communication: Lessons from the SHIM Project. In: Lee, Sunggu (Hrsg.) ; Narasimhan, Priya (Hrsg.): *Software Technologies for Embedded and Ubiquitous Systems* Bd. 5860. Springer Berlin / Heidelberg, 2009, S. 276–287. – 10.1007/978-3-642-10265-3_25

[22] Edwards, Stephen A.: Tutorial: Compiling Concurrent Languages for Sequential Processors. In: *ACM Transactions on Design Automation of Electronic Systems* 8 (2003), April, Nr. 2, S. 141–187

[23] Edwards, Stephen A. ; Kim, Sungjun ; Lee, Edward A. ; Liu, Isaac ; Patel, Hiren D. ; Schoeberl, Martin: A Disruptive Computer Design Idea: Architectures with Repeatable Timing. In: *Proceedings of IEEE International Conference on Computer Design (ICCD)* IEEE (Veranst.), URL `http://chess.eecs.berkeley.edu/pubs/614.html`, October 2009. – Lake Tahoe, CA

[24] Gaisler, Aeroflex: *GRLIB IP Core User's Manual Version 1.0.22*. Aeroflex Gaisler AB, Kungsgatan 12, 411 19 Göteborg, SE: , 4 2010. – `http://gaisler.com/products/grlib/grip.pdf`

[25] Gaisler, Jiri: A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture. In: *International Conference on Dependable Systems and Networks, 2002. DSN 2002. Proceedings.*, 2002, S. 409 – 415

[26] Garner, R.B. ; Agrawal, A. ; Briggs, F. ; Brown, E.W. ; Hough, D. ; Joy, B. ; Kleiman, S. ; Muchnick, S. ; Namjoo, M. ; Patterson, D. ; Pendleton, J. ; Tuck, R.: The Scalable Processor Architecture (SPARC). In: *Compcon Spring '88. Thirty-Third IEEE Computer Society International Conference, Digest of Papers*, Februar 1988, S. 278 –283

[27] Girault, Alain: A Survey of Automatic Distribution Method for Synchronous Programs. In: Maraninchi, F. (Hrsg.) ; Pouzet, M. (Hrsg.) ; Roy, V. (Hrsg.): *International Workshop on Synchronous Languages, Applications and Programs (SLAP '05)*. Edinburgh, UK : Elsevier Science, April 2005 (Electronic Notes in Theoretical Computer Science)

[28] Girault, Alain ; Ménier, Clément: Automatic Production of Globally Asynchronous Locally Synchronous Systems. In: Sangiovanni-Vincentelli, Alberto (Hrsg.) ; Sifakis, Joseph (Hrsg.): *Embedded Software* Bd. 2491. Springer Berlin / Heidelberg, 2002, S. 266–281

[29] Girault, Alain ; Nicollin, Xavier ; Pouzet, Marc: Automatic Rate Desynchronization of Embedded Reactive Programs. In: *ACM Transactions on Embedded Computing Systems* 5 (2006), Nr. 3, S. 687–717. – ISSN 1539-9087

[30] Goodacre, John ; Sloss, Andrew N.: Parallelism and the ARM Instruction Set Architecture. In: *Computer* 38 (2005), Juli, Nr. 7, S. 42 – 50. – ISSN 0018-9162

[31] Graham, Susan L. ; Kessler, Peter B. ; Mckusick, Marshall K.: Gprof: A Call Graph Execution Profiler. In: *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*. New York, NY, USA : ACM, 1982, S. 120–126. – ISBN 0-89791-074-5

[32] Halbwachs, Nicolas ; Caspi, Paul ; Raymond, Pascal ; Pilaud, Daniel: The Synchronous Data-Flow Programming Language LUSTRE. In: *Proceedings of the IEEE* 79 (1991), September, Nr. 9, S. 1305–1320

[33] Harel, D. ; Pnueli, A.: On the Development of Reactive Systems. In: *Logics and Models of Concurrent Systems* (1985), S. 477–498. ISBN 0-387-15181-8

[34] Harel, David: Statecharts: A Visual Formalism for Complex Systems. In: *Science of Computer Programming* 8 (1987), Juni, Nr. 3, S. 231–274

[35] Herlihy, Maurice: The Multicore Revolution. In: Arvind, V. (Hrsg.) ; Prasad, Sanjiva (Hrsg.): *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science* Bd. 4855. Springer Berlin / Heidelberg, 2007, S. 1–8

[36] Ip, Nicholas ; Edwards, Stephen: A Processor Extension for Cycle-Accurate Real-Time Software. In: Sha, Edwin (Hrsg.) ; Han, Sung-Kook (Hrsg.) ; Xu, Cheng-Zhong (Hrsg.) ; Kim, Moon (Hrsg.) ; Yang, Laurence (Hrsg.) ; Xiao, Bin (Hrsg.): *Embedded and Ubiquitous Computing* Bd. 4096. Springer Berlin / Heidelberg, 2006, S. 449–458. – 10.1007/11802167_46

[37] Karl, Wolfgang: Vorlesung: Mikroprozessoren. Karlsruhe, Germany, 2006. – `http://capp.itec.kit.edu/teaching/mp/ss06/mp06-10.pdf`

[38] Kirner, Raimund ; Puschner, Peter: Obstacles in Worst-Case Execution Time Analysis. In: *ISORC '08: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*. Washington, DC, USA : IEEE Computer Society, 2008, S. 333–339. – ISBN 978-0-7695-3132-8

[39] Kistler, M. ; Perrone, M. ; Petrini, F.: Cell Multiprocessor Communication Network: Built for Speed. In: *Micro, IEEE* 26 (2006), Mai, Nr. 3, S. 10 –23. – ISSN 0272-1732

[40] Le Guernic, P. ; Benveniste, A. ; Bournai, P. ; Gautier, T.: Signal–A Data Flow-Oriented Language for Signal Processing. In: *Acoustics, Speech and Signal Processing, IEEE Transactions on* 34 (1986), apr., Nr. 2, S. 362 – 374. – ISSN 0096-3518

[41] Lee, Edward A. ; Edwards, Stephen A. ; Kim, Sungjun ; Patel, Hiren D. ; Schoeberl, Martin: Reconciling Repeatable Timing with Pipelining and Memory Hierarchy. In: *Workshop on Reconciling Performance with Predictability (RePP), Grenoble, France*, Oktober 2009. – Columbia University, New York, NY, USA, 2009 http://www1.cs.columbia.edu/~sedwards/papers/edwards2009reconciling.pdf

[42] Li, Xin ; Boldt, Marian ; von Hanxleden, Reinhard: Mapping Esterel onto a Multi-Threaded Embedded Processor. In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*. San Jose, CA, October 21–25 2006

[43] Lickly, Ben ; Liu, Isaac ; Kim, Sungjun ; Patel, Hiren D. ; Edwards, Stephen A. ; Lee, Edward A.: Predictable Programming on a Precision Timed Architecture. In: *Proceedings of Compilers, Architectures, and Synthesis of Embedded Systems (CASES'08)*. Atlanta, USA, Oktober 2008

[44] Lickly, Ben ; Liu, Isaac ; Kim, Sungjun ; Patel, Hiren D. ; Edwards, Stephen A. ; Lee, Edward A.: Predictable Programming on a Precision Timed Architecture. In: *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*. Atlanta, GA, USA : ACM, 2008, S. 137–146. – ISBN 978-1-60558-469-0

[45] Lynch, Nancy A.: *Distributed Algorithms (The Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, 1996. – ISBN 1558603484

[46] Martin, Jeff: *Propeller Manual Version 1.1*. Parallax Inc., 599 Menlo Drive, Rocklin, California 95765, USA: , 2009. – http://www.parallax.com/Portals/0/Downloads/docs/prod/prop/WebPM-v1.1.pdf

[47] May, David: *The XMOS XS1 Architecture*. XMOS Ltd., Venturers House, King Street, Bristol, BS1 4PB, UK: , 10 2009. – http://www.xmos.com/published/xmos-xs1-architecture-0?ver=xs1_en.pdf

[48] May, David ; Dixon, Ali ; Oung, Ayewin ; Muller, Henk: *XS1 System Specification (Version 1.3)*. XMOS Ltd. Venturers House, King Street, Bristol, BS1 4PB, UK: , 1 2009. – http://www.xmos.com/published/xs1-g-system-specification

[49] Muttersbach, J. ; Villiger, T. ; Fichtner, W.: Practical Design of Globally-Asynchronous Locally-Synchronous Systems. In: *Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems, 2000. (ASYNC 2000)*, 2000, S. 52 –59

[50] Pham, D.C. ; Aipperspach, T. ; Boerstler, D. ; Bolliger, M. ; Chaudhry, R. ; Cox, D. ; Harvey, P. ; Harvey, P.M. ; Hofstee, H.P. ; Johns, C. ; Kahle, J. ; Kameyama, A. ; Keaty, J. ; Masubuchi, Y. ; Pham,

M. ; Pille, J. ; Posluszny, S. ; Riley, M. ; Stasiak, D.L. ; Suzuoki, M. ; Takahashi, O. ; Warnock, J. ; Weitzel, S. ; Wendel, D. ; Yazawa, K.: Overview of the Architecture, Circuit Design, and Physical Implementation of a First-Generation Cell Processor. In: *Solid-State Circuits, IEEE Journal of* 41 (2006), Januar, Nr. 1, S. 179 – 196. – ISSN 0018-9200

[51] Potop-Butucaru, D. ; Simone, R. de ; Talpin, J.-P.: The Synchronous Hypothesis and Synchronous Languages. In: Zurawski, R. (Hrsg.): *Embedded Systems Handbook*. CRC Press, 2005

[52] Potop-Butucaru, Dumitru ; Caillaud, Benoît ; Benveniste, Albert: Concurrency in Synchronous Systems. In: *Formal Methods in System Design* 28 (2006), S. 111–130. – ISSN 0925-9856

[53] Roop, Partha S. ; Andalam, Sidharta ; Hanxleden, Reinhard von ; Yuan, Simon ; Traulsen, Claus: Tight WCRT Analysis for Synchronous C Programs. In: *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'09)*. Grenoble, France, Oktober 2009

[54] Sorel, Y.: Massively Parallel Computing Systems with Real Time Constraints: The „Algorithm Architecture Adequation" Methodology. In: *Massively Parallel Computing Systems, 1994., Proceedings of the First International Conference on*, Mai 1994, S. 44 –53

[55] Starke, Falk ; Traulsen, Claus ; Hanxleden, Reinhard von: Executing Safe State Machines on a Reactive Processor / Christian-Albrechts-Universität Kiel, Department of Computer Science, Kiel, Germany. März 2009 (0907). – Technical Report

[56] Stevens, W. R.: *Programmieren von UNIX-Netzwerken*. Carl Hanser Verlag, 2000. – 2. Auflage. – ISBN 3446213341

[57] Thomas, David B. ; Howes, Lee ; Luk, Wayne: A Comparison of CPUs, GPUs, FPGAs, and Massively Parallel Processor Arrays for Random Number Generation. In: *FPGA '09: Proceeding of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. New York, NY, USA : ACM, 2009, S. 63–72. – ISBN 978-1-60558-410-2

[58] Vasudevan, Nalini ; Edwards, Stephen A.: Celling SHIM: Compiling Deterministic Concurrency to a Heterogeneous Multicore. In: *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*. New York, NY, USA : ACM, 2009, S. 1626–1631. – ISBN 978-1-60558-166-8

[59] Vasudevan, Nalini ; Edwards, Stephen A.: A Determinizing Compiler. In: *Proceedings of Program Language Design and Implementation (pldi)*, 2009

[60] von Hanxleden, Reinhard: *Lectures: Model-Based Design and Distributed Real-Time Systems*. Christian-Albrechts-Universität zu Kiel, Department of Computer Science. 2009. – http://rtsys.informatik.uni-kiel.de/teaching/ss09/v-model/lectures/

[61] von Hanxleden, Reinhard: *Lectures: Synchronous Languages*. Christian-Albrechts-Universität zu Kiel, Department of Computer Science. 2009. – http://rtsys.informatik.uni-kiel.de/teaching/09ws/v-synch/lectures/

[62] von Hanxleden, Reinhard: SyncCharts in C / Christian-Albrechts-Universität zu Kiel, Department of Computer Science. Mai 2009 (0910). – Technical Report

[63] von Hanxleden, Reinhard: *Lectures: Embedded Real-Time Systems*. Christian-Albrechts-Universität zu Kiel, Department of Computer Science. 2010. – http://rtsys.informatik.uni-kiel.de/teaching/10ss/v-emb-rt/lectures/

[64] Wikipedia: *MIPS-Architektur — Wikipedia, Die freie Enzyklopädie*. – [Stand 30. September 2010] http://de.wikipedia.org/w/index.php?title=MIPS-Architektur&oldid=78405530

[65] Woo, Dong H. ; Lee, Hsien-Hsin S.: Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era. In: *Computer* 41 (2008), S. 24–31. – ISSN 0018-9162

[66] Yoong, Li H. ; Roop, Partha ; Salcic, Zoran ; Gruian, Flavius: Compiling Esterel for Distributed Execution. In: *SLAP '06: International Workshop on Synchronous Languages, Applications, and Programming*. Vienna, Austria, März 2006

A. Repository-Übersicht

fft/ enthält Code der schnellen Fourier-Transformation aus dem Benchmark SciMark 2.0. Um Laufzeitmessungen zu ermöglichen, wurde ein kleines Programm ergänzt, das in mehreren parallelen Prozessen rechnet.

pret_iss/ enthält den entpackten Code des PRET-Simulators. Er muss wie in Anhang B beschrieben kompiliert werden.

SciMark2/ enthält den unveränderten Code von SciMark 2.0.

scmc-linux/ enthält die Linux-Implementierung von SCmc, beschrieben in Abschnitt 6.7.2 und Anhang C.

scmc-pret/abro-singlethread/ enthält SC-Code ohne Veränderungen zur Ausführung mit nur einem Hardware-Thread.

scmc-pret/blockingloc/ enthält die PRET-Implementierung von SCmc mit Certainty-Matrix und blockierenden Locations, beschrieben in Abschnitt 6.6.4 und Anhang APret.

scmc-pret/priobarrier/ enthält die PRET-Implementierung von SCmc mit Priority-Barriers, beschrieben in Abschnitt 6.7.1 und Anhang APret.

scmc-pret/fifo_ringbuffer/ enthält den experimentellen Ringpuffer, der Am Ende von Abschnitt 6.7.2 erwähnt wurde.

sc-r3317-benchmark/ enthält SC, zu Benchmarking-Zwecken erweitert um eine Schleife in der `main`-Funktion und mit kleineren Modifikationen im Makefile.

Alle PRET-Implementierungen enthalten abgewandelte Versionen von ABRO.

B. Benutzung des PRET-Simulators

Der Source-Code des PRET-Simulators kann als Tarball heruntergeladen und entpackt[1] oder aber mit
`svn co svn://source.eecs.berkeley.edu/chess/pret/trunk/pret_iss` aus dem Subversion-Repository bezogen werden.

- Nach dem Entpacken ist die Umgebungsvariable `$PRET_ISS` auf das Verzeichnis zu setzen, in dem sich der PRET-Code befindet.
- Wurde der Code aus dem Repository bezogen, muss `bootstrap.sh` ausgeführt werden.
- Mit `./configure` und anschließendem `make` wird der Simulator kompiliert.
- Sollte die Version aus dem Tarball nicht kompilieren, muss eventuell `#include <stdint.h>` in `include/instruction.h` eingefügt werden.

Für diese Arbeit wurden beide Versionen getestet, aber nur die aus dem Repository verwendet.

Der Simulator bringt ein Python-Script zum Kompilieren der – in der Regel als getrennte C-Dateien abgelegten – Programme für die Hardware-Threads mit[2]:

- Mit `scripts/compile_threads.py tests/tests/pret/sys/smile/smile/` kann zum Beispiel das „Smile“-Beispiel kompiliert werden.
- `${PRET_ISS}/src/pret tests/tests/pret/sys/smile/smile/ -1`
  führt das Beispiel aus. Der zweite Parameter gibt die Anzahl der zu simulierenden Zyklen an, `-1` bedeutet unendlich.

Listing B.1 zeigt die vermengte Ausgabe des „Smile“-Beispiels unter Linux.

[1] `http://chess.eecs.berkeley.edu/pret/src/pret-1.0/pret-1.0.tar.gz`

[2] `http://chess.eecs.berkeley.edu/pret/src/pret-1.0/pret_simulator.html#[[Compiling%20Programs]]`

```
1 ------------------------------------------------
2         Start Simulation
3 ------------------------------------------------
4   (    ||| /   o    o  _oo |_ o  _oo     \     |||  )
5
6
7
8
9    (    | | \/   o     o\  \oo_ |_ o/  _oo   _  /\ /    | |   )
```

Listing B.1: Ausgabe des „Smile“-Beispiels aus dem PRET-Simulator-Paket

Für eigene Programme muss ein Verzeichnis angelegt werden, das die folgenden Dateien enthält:

`thread[0-5].c` enthalten die Programme, die als einzelne Threads ausgeführt werden sollen.

`thread[0-5].makefile` sind die zugehörigen Makefiles.

`thread_common.makefile` enthält allen Threads gemeinsame Regeln.

In den Verzeichnissen mit den beiden SCmc-Implementierungen wurde außerdem ein allgemeines Makefile angelegt:

`make` erzeugt alle sechs Programme mit Hilfe des oben genannten Python-Scripts.

`make run` führt den Simulator mit den Programmen im lokalen Verzeichnis aus.

`make regression` kompiliert erneut und führt dann ebenfalls den Simulator aus.

Die Dateien `thread[0-5].[dspm|ispm]` enthalten die Adressbereiche, die in den Daten- (`.dspm`) und Instruktionsteil (`.ispm`) des Scratchpad-RAMs geladen werden sollen.

C. Benutzung der Linux-Implementierung

Im Verzeichnis der Linux-Implementierung befinden sich einige Scripte, die SCmc-Dateien und Programme, die jeweils auf mehrere C-Dateien für die verschiedenen Locations aufgeteilt sind. Die Aufteilung erfolgt durch Benennung der Dateien mit Nummer der Location (0-9) am Ende. Im Makefile sind die Programme als Targets definiert, die C-Dateien für die einzelnen Locations werden automatisch gefunden und kompiliert:

`make abro` erzeugt zum Beispiel die ausführbaren Dateien `abro0` und `abro1`, weil `abro0.c` und `abro1.c` im Verzeichnis gefunden werden.

`make run` kompiliert `abro0` und `abro1` erneut und führt sie aus. Die Ausgabe von `abro1` wird unterdrückt.

`make regression` kompiliert alle Programme erneut, führt eine Auswahl von ihnen aus und bricht ab, falls ein Fehler aufgetreten ist.

Die Scripte dienen der einfachen Ausführung oder Zeitmessung:

- `start.sh` erwartet als ersten Parameter den Namen eines Programms und führt alle zugehörigen Location-Programme im Verzeichnis aus. Die Ausgabe aller Locations außer 0 wird unterdrückt.
- `benchmark.sh` erwartet als zweiten Parameter eine Zahl, die angibt, wie oft die Ausführung der Programme wiederholt werden soll. Die Rechenzeit, die die Location-Programme benötigen, wird gemessen und ausgegeben. Sämtliche Ausgaben werden nach `/dev/null` umgeleitet.

Die Location-Programme erkennen ihre Nummer selbst, entweder an ihrem Namen oder einem Kommandozeilenparameter.

D. Beschreiben von FPGAs

Um einen IP-Core wie den LEON3 auf einen FPGA zu schreiben, benötigt man eine JTAG-Verbindung zu diesem. Unter Linux wird für den USB-JTAG-Adapter von Xilinx je nach Version der Software ein USB-Treiber benötigt[1].

Für einige FPGAs genügt die Software „ISE WebPack“ von Xilinx[2], für manche wird die Vollversion benötigt.

Die IP-Bibliothek *GRLIB* von Aeroflex Gaisler[3] enthält vorbereitete Designs für verschiedene FPGA-Boards, zum Beispiel für ein Board mit einem Spartan3-1500 im Verzeichnis `designs/leon3-gr-xc3s-1500/`.

Mit `make xconfig` konfiguriert man das Design. Abbildung D.1 zeigt die Oberfläche des Konfigurationstools.

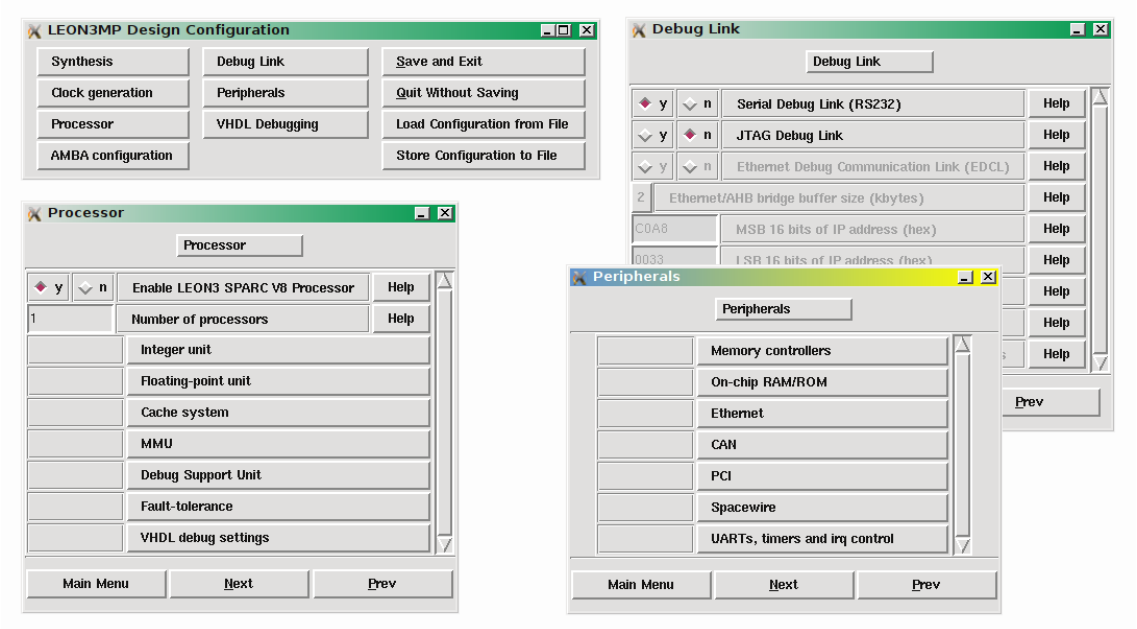


Abbildung D.1.: Konfigurationstool der GRLIB

Zur Erzeugung eines `bin`-Files wie in Abschnitt 4.5 beschrieben muss der Pfad zu den ISE-Executables in der entsprechenden Umgebungsvariable gesetzt sein.

[1] Siehe `http://www.rmdir.de/~michael/xilinx/`, `http://git.zerfleddert.de/cgi-bin/gitweb.cgi/usb-driver?a=blob_plain;f=README;hb=HEAD`, sowie „Jungo WinDriver“: `http://www.jungo.com/st/do/download_new.php?product=WinDriver`

[2] Installationsanleitung: `http://wiki.ubuntuusers.de/ISE_WebPACK`

[3] `http://www.gaisler.com/cms/index.php?option=com_content&task=section&id=13&Itemid=125`

Mit `make ise` wird die Erzeugung gestartet. Zur Übertragung auf das Board wird das Programm `impact` aus dem ISE-Programmpaket verwendet.

Um zu überprüfen, ob sich ein funktionsfähiger Prozessor auf dem FPGA befindet, wird dieser über das JTAG-Kabel mit dem Debugger *GRMON*[4] kontaktiert.

Für diese Arbeit standen zwei Boards mit FPGAs von Xilinx zur Verfügung, eins mit einem Virtex-4 FX12, eins mit einem Spartan-3 1500. Versuche, einen LEON3 funktionsfähig auf einen der FPGAs zu schreiben, schlugen fehl.

Ein Dualcore-System ist bereits zu groß für den Virtex-4 FX12, der einen PowerPC-405-Kern, aber nur 12312 Logic Cells enthält[5]. Ein Spartan-3 1500 ist mit 29952 Logic Cells[6] groß genug, ein System mit drei LEON3-Kernen aufzunehmen, das Board verweigerte aber aus unbekannten Gründen den Kontakt.

Die Listings D.1 und D.2 zeigen die Auslastung der beiden FPGAs laut Ausgabe der Software ISE beim Erzeugen des jeweiligen `bin`-Files.

```
1  Number of Slices:                     7425  out of   5472   135% (*)
2  Number of Slice Flip Flops:           4064  out of  10944    37%
3  Number of 4 input LUTs:              13128  out of  10944   119% (*)
4     Number used as logic:             13080
5     Number used as Shift registers:      48
6  Number of IOs:                         199
7  Number of bonded IOBs:                 169  out of    320    52%
8     IOB Flip Flops:                     202
9  Number of FIFO16/RAMB16s:               14  out of     36    38%
10    Number used as RAMB16s:              14
11 Number of GCLKs:                         9  out of     32    28%
12 Number of DCM_ADVs:                      4  out of      4   100%
```

Listing D.1: Auslastung des Virtex-4-FX12 mit zwei LEON3-Kernen

```
1  Number of Slices:                    10746  out of  13312    80%
2  Number of Slice Flip Flops:           5663  out of  26624    21%
3  Number of 4 input LUTs:              20398  out of  26624    76%
4     Number used as logic:             20330
5     Number used as Shift registers:      68
6  Number of IOs:                         265
7  Number of bonded IOBs:                 157  out of    333    47%
8     IOB Flip Flops:                     213
9  Number of BRAMs:                         6  out of     32    18%
10 Number of MULT18X18s:                   12  out of     32    37%
11 Number of GCLKs:                         2  out of      8    25%
12 Number of DCMs:                          2  out of      4    50%
```

Listing D.2: Auslastung des Spartan-3-1500 mit drei LEON3-Kernen

[4] http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=190&Itemid=124

[5] http://www.em.avnet.com/evk/home/0,1707,RID%253D0%2526CID%253D25424%2526CCD%253DUSA%2526SID%253D32214%2526DID%253DDF2%2526LID%253D32232%2526PRT%253D0%2526PVW%253D%2526BID%253DDF2%2526CTP%253DEVK,00.html

[6] http://www.protel.com/products/nanoboard_nb2/daughter-boards/de/xilinx-spartan-3.cfm