Hardware Operating Systems

Ulrich Langenbach
basseuph@cs.tu-berlin.de

Dynamic Reconfigurable Embedded Systems
PDV Seminar SS06

Zusammenfassung

Eingebettete Systeme werden immer vielfältiger eingesetzt, die Anforderungen steigen insbesondere an Flexibilität und Leistungsreserve. Aus der aus diesen Anforderungen entstehenden Erweiterung eingebetteter Systeme um dynamisch rekonfigurierbare Einheiten, folgt der Bedarf an Betriebssystemen, die auch diese Einheiten kontrollieren und abstrahieren. Es werden grundsätzliche Bestandteile dieser Systeme erläutert und einige verschiedene Herangehensweisen an Hand von Beispielen erläutert.

Abstract

The growing wideley use of embedded systems leads to more flexible high performance systems. To fulfill this goal of powerful systems, dynamic reconfigurable logicblocks have been added to traditional embedded systems. As an answer to the need of abstraction and control of this dynamic reconfigurable logic units hardware operating systems are created. In this article several basic subsystems are discussed and different approaches in implementation are shown by examples.

1 Einführung

Moderne eingebettete Systeme sollen immer komplexeren Anforderungen, wie zum Beispiel Multimedia- oder Netzwerkanwendungen, gerecht werden. Gleichzeitig werden die Anforderungen vielfältiger, so sollen diese Systeme bspw. aufwändige Verschlüsselungsalgorithmen unterstützen. Die Kombination aus dem Bedarf erhöhter Flexibilität und Leistungsfähigkeit führt zu der Integration dynamisch rekonfigurierbarer Einheiten (auch Hardware Tasks oder nur Task genannt) in das Gesamtsystem, zum Beispiel mit FPGAs. Nun müssen diese Strukturen und Einheiten auch verwaltet werden, weshalb sich nun der Forschungsbereich der Hardware Operating Systems abzeichnet. Außerdem führen diese Systeme zu einem höheren Abstraktionsniveau als bei herkömmlichen ASICs (Application Specific Integrated Circuits), welche im Gesamtsystem meist als Hauptprozessor (ARM, MIPS) mit Spezial-Co-Prozessoren ausgeführt sind. Diese Arbeit wird im ersten Teil erläutern wo die Unterschiede und Gemeinsamkeiten zwischen traditionellen Betriebssystemen und Hardware-Betriebssystemen liegen. Dazu wird auf grundlegende Elemente eines Hardware- Betriebssystemen eingegangen und dann verschiedene Herangehensweisen an die Implementierung mit Beispielen erläutert.

1.1 Motivation

Als erstes stellt sich die Frage, warum man eigentlich ein Hardware- Betriebssystem braucht. Ein Hardware-Betriebssystem sorgt für die notwendige Abstraktion von Hardware- Details übernimmt und die Verwaltung derselben. Außerdem muss, wenn ein dynamisch rekonfigurierbares System vorliegt, auch eine Rekonfiguration ausgelöst und überwacht werden. Neben und aus diesen Gründen, ergeben sich weitere, noch wichtigere Aspekte, die für die Einführung eines solchen Systems sprechen. So wird die Portabilität eines Systems deutlich gesteigert, wenn nur noch eine Basis, das Hardware- Betriebssystem, und nicht mehr das Gesamtsystem von einer Plattform auf eine andere portiert werden muss. Ein weiterer Vorteil ergibt sich daraus, dass ein komplexes System aus verschiedenen Subsystemen besteht, welches im Zuge der (Weiter-) Entwicklung auch notwendigerweise einmal umstrukturiert wird. Dazu gehört auch die Verschiebung von Funktionen von Hardware in Softwarekomponenten und umgekehrt. Auch diese Systemrepartitionierung wird einfa-

cher, da nun allgemeinere Zwischenschichten zur Verfügung stehen. Aus klarerer Abgrenzung und einfacheren, allgemeinen Schnittstellen zu Subsystemen ergibt sich auch ein weniger komplexes Debuggen von einzelnen Komponenten. Aus den aufgezählten Gründen und dadurch, dass dieselbe Hardware in einem dynamisch rekonfigurierbaren System über die Zeit betrachtet deutlich effektiver genutzt wird, resultiert letztendlich eine höhere Produktivität und damit eine größere Wirtschaftlichkeit des Gesamtsystems, bei gleichzeitig gesteigerter Leistung und Flexibilität.

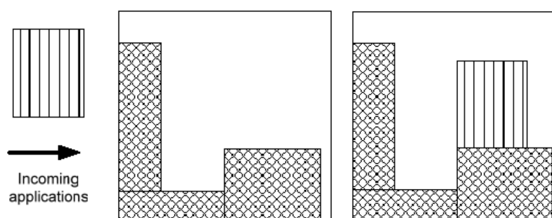

Abbildung 1: Platzieren eines Tasks in ein System [Wig05]

1.2 Aufgaben eines Betriebssystems

Die folgende Liste spiegelt die Hauptaufgaben eines traditionellen Betriebssystems wieder. Alle Punkte dieser Liste sind, wenn vielleicht auch in einfacher Implementation in jedem Betriebssystem enthalten.

- Speicherverwaltung
- Prozessverwaltung
- Scheduling
- Hardwareverwaltung
- Zeitdienste
- Synchronisationsdienste
- Kommunikation
- Abstraktion

Ein Hardware- Betriebssystem dagegen hat nachstehende Aufgaben, wobei es meist natürlich Teil eines Gesamtsystems ist, in dem es als Erweiterung der Hardwareverwaltung eines traditionellen Systems gesehen werden kann. Allerdings gibt es auch andere Ansätze, welche später am Beispiel vom Hybrid THREAD Projekt [DA05] betrachtet werden.

⇒ Platzierung

⇒ Flächenverwaltung im FPGA

⇒ (Partitionierung der rekonfigurierbaren Hardware)

⇒ (Routing)

⇒ Kommunikation

⇒ auch Spezialblöcke

- Scheduling
- Synchronisation
- Rekonfiguration

Auf die mit Pfeilen gekennzeichneten Punkte wird im folgenden Teil näher eingegangen. Einige Punkte sind eingeklammert, da diese bisher nur in wenigen Projekten Teil eines Hardware- Betriebssystems geworden sind, was im wesentlichen an den technischen Einschränkungen bei der partiellen Rekonfiguration von FPGAs (Field Programmable Gate Array) liegt.

2 Aufbau

2.1 Platzierung

Bei der Platzierung als Teil eines Hardware- Betriebssystems gibt es zwei unterschiedliche Aufgaben. Zum einen findet eine Platzierung eines Tasks, also eines Hardwaremoduls, auf höherer Ebene in der Fläche der rekonfigurierbaren Einheit statt. Zum Anderen muss dieser Task auch intern platziert sein, sprich seine Logik schon geometrisch beschrieben sein. Wir beschäftigen uns nun hier nur mit der Platzierung eines Tasks im dynamisch rekonfigurierbaren System, da die Tasks im Allgemeinen schon fertig synthetisiert vorliegen. Wie in Abbildung 1 gezeigt, wird nun ein Task in einem System platziert. Es sind dabei schon drei andere Funktionsblöcke vorhanden. Dieses Problem ist NP hart und erfordert somit Heuristiken, damit es in vernünftiger Laufzeit gelöst werden kann. Das Problem ähnelt dem Rucksackproblem (engl. bin packing problem), welches auch aus dem Bereich Hochleistungsrechnen mit Clustern bekannt ist, für das es daher diverse Algorithmen mit heuristischen Verfahren, wie zum Beispiel genetische Algorithmen und Simulated Annealing, gibt.

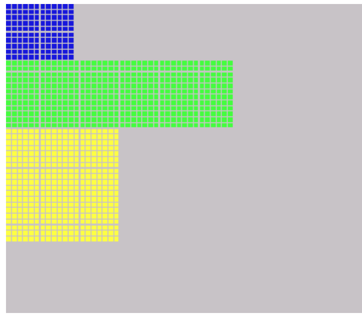

Abbildung 2: Von Tasks belegte Flächen im FPGA [Wig05]

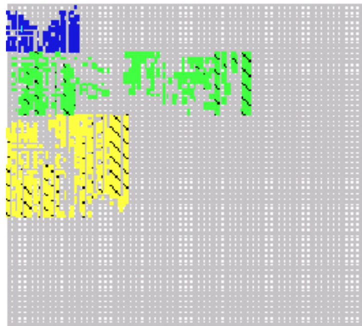

Abbildung 3: Tatsächlich belegte Logikblöcke von Tasks im FPGA [Wig05]

2.2 Fragmentierung

Ein Algorithmus zur Platzierung soll externe Fragmentierung vermeiden, um den Platz in der rekonfigurierbaren Einheit effektiv zu nutzen. Was das bedeutet kann man anhand Abbildung 2 sehen. So erkennt man zum einen die Zeilen und Spaltenstruktur eines FPGAs, zum anderen auch die verschiedenen, unterschiedlich eingefärbten, Tasks. Externe Fragmentierung ist nun der Bereich der Fläche, der nicht durch einen anderen Task genutzt werden kann, da zum Beispiel, die Fläche zu klein ist. Ähnliches könnte etwa oberhalb des grünen, neben dem blauen Task passieren oder unter dem Gelben.

Interne Fragmentierung entsteht innerhalb eines Tasks zum Beispiel dadurch, dass gewisse Speicher und Pads benutzt werden müssen, die räumlich nicht so dicht beieinander liegen. Oder es entstehen Spaltenstrukturen durch die Benutzung von Carry Leitungen. Jedenfalls kann man dem Abhilfe schaffen, indem man die Grundgeometrie eines Tasks auf Polygone erweitert, da dieses jedoch die Platzierung eines Tasks deutlich erschwert, verzichtet man im Allgemeinen darauf und nimmt damit interne Fragmentierung in Kauf.

2.3 Taskpartitionierung

Eine weitere Möglichkeit interne und externe Fragmentierung zu reduzieren kann man nutzen, wenn ein Task vom Betriebssystem noch unterteilbar ist. Das Vorgehen mit Zusammenfassung einiger Operationen und der folgenden Einteilung in verschiedene Partitionen ist in Abbildung 4 verdeutlicht. Dabei wird die gesamte Netzliste in Teile untergliedert, welche separat platziert und verdrahtet werden. Um diese Aufteilung durch das Betriebssystem zu ermöglichen, muß der Task in Form eines SDF (engl. Synchronous Dataflow Graph, synchroner Datenflußgraph) vorliegen, damit das Betriebssystem die nötigen Informationen hat. Im SDF sieht man parallel laufende Prozessschritte nebeneinander in der Horizontalen angeordnet. Den zeitlichen Verlauf ergibt sich aus der vertikalen Anordnung. Wenn so eine Taskpartitionierung vorgenommen werden soll, um zum Beispiel Fragmentierung zu vermeiden oder in Gänze nicht in den programmierbaren Bereich passende Tasks doch, zeitlich gestaffelt, ausführen zu können, setzt das voraus, dass zum einen ein Mechanismus zum Platzieren im System vorhanden ist und zum Anderen zur Laufzeit ein Routing von Leitungen im programmierbaren Bereich möglich ist.

2.4 Routing

Routing stellt an ein Hardware- Betriebssystem ebenso wie das Platzieren eine hohe Anforderung bezüglich Rechenaufwand und Speicherbedarf, denn es ist auch ein NP hartes Problem, dem daher meist mit Heuristiken begegnet wird. Dieses Problem ist in diesem Fall

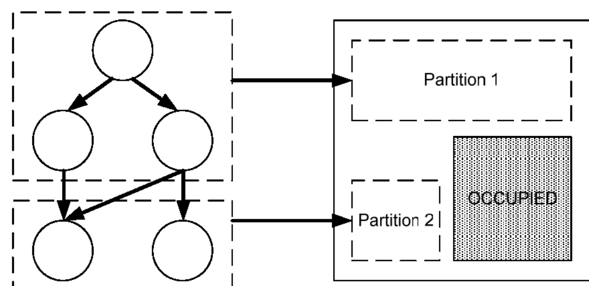

Abbildung 4: Taskpartitionierung [Wig05]

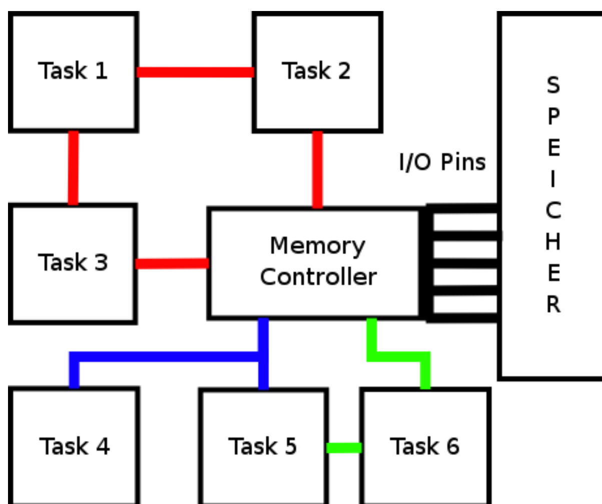

Abbildung 5: Verschiedene Kommunikationsstrukturen in der Hardware [Wig05]

allerdings oft gar kein so Großes, da vornehmlich lokale Beziehungen unter den Funktionsblöcken bestehen. Das ermöglicht meist eine schnelle Bearbeitung des Problems mit einfacheren Algorithmen. Weiterhin schränkt die Struktur der verwendeten programmierbaren Logikeinheit die Komplexität der Möglichkeiten ein, da zum Beispiel im Gegensatz zu pcb routing (engl. printed circuit board, Platine) im FPGA nur horizontale und vertikale Verbindungen möglich sind. Trotzdem wird von der Möglichkeit des Online-Routings nur selten Gebrauch gemacht, da auch dabei die technischen Einschränkungen der partiellen Rekonfiguration in FPGAs dieses nicht umfassend ermöglichen. Es wurde allerdings in einem System implementiert, welches immer einen kompletten FPGA rekonfigurierte [Wig05].

2.5 Kommunikationsstrukturen

Ein weiteres wichtiges Merkmal eines jeden Rechensystems ist die Bustopologie. Sie bestimmt in starkem Maße die Kommunikationsstrukturen der Hardware. Das gilt natürlich auch für ein dynamisch rekonfigurierbares System. Aus diesem Grund möchte ich kurz auf die verschiedenen Netztopologien eingehen, die man in den Systemen findet.

Die einfachste Struktur ist die direkte Verbindung von Funktionsblöcken über jeweils spezielle Interfaces. Zu sehen ist diese Struktur in Abbildung 5 grün gezeichnet. Mit der direkten Verbindung erhält man die größte Parallelität im Kommunikationssystem, allerdings ist die Struktur unflexibel. Daher benötigt sie in einem dynamisch rekonfigurierbaren System ein dynamisches Routing, damit die Verbindungen auch bei Ortsveränderungen noch hergestellt werden können.

Da das in vielen Systemen nicht möglich ist, weicht man dann auf die nächst kompliziertere Möglichkeit, den Bus, aus. In Abbildung 5 ist der Bus blau gekennzeichnet. Der Bus ist tatsächlich die am weitesten verbreitete Topologie, wobei sie in den PCs zum Teil von geschalteten seriellen Punkt zu Punkt Verbindungen verdrängt wird. Der Bus bietet eine einheitliche Schnittstelle über die alle Komponenten miteinander kommunizieren können. Allerdings kann auf dieses flexible Kommunikationsmedium zu einem Zeitpunkt nur eine Komponente schreiben. Durch diese geringe Parallelität können schnell Engpässe und Blockaden entstehen.

Um in Systemen mit erhöhten Datenraten auf den nötigen Durchsatz zu kommen, kann man auf eine Pipeline- Architektur ausweichen, welche wie der Bus ein einheitliches Interface, dabei jedoch eine hohe Parallelität bietet. In Abbildung 5 ist die Pipeline rot dargestellt. Eine solche Topologie eignet sich gut für datenstromorientierte Systeme in Audio-, Video- und Netzwerkanwendungen, da die Funktionen, die nacheinander auf einen Datenfluss angewendet werden, in eben dieser Reihenfolge hintereinander geschaltet werden können. Wird eine erhöhte Flexibilität benötigt, kann dynamisches Routing für Datenflüsse eingeführt werden, was dann letztendlich zu einem Network on Chip (NoC) führt.

Übernimmt der Memory Controller die Funktion eines Switches, so so stellt das Gesamtsystem in Abbildung 5 einen Network On Chip (NoC) dar. So kann dann Task 5 über diesen Switch mit Task 2 kommunizieren. Meist arbeiten NoCs auch paketorientiert und bieten damit ähnliche Möglichkeiten wie herkömmliche Rechnernetzwerke, deren Prinzipien gut verstanden sind. Das Problem bei diesem Vorgehen ergibt sich allerdings aus dem doch zum Teil beachtlichen Overhead, bestehend aus Headern der Pakete. Zusätzlich ergibt sich eine Latenz durch das Verschicken eines Paketes über Switches und Router sowie einen erhöhter Platzbedarf auf dem Chip, um Switches unterzubringen. Trotz dieser Nachteile bieten NoCs durchaus Potenziale für neue Entwicklungen in den schon genannten Einsatzgebieten, wobei noch Bedarf an neuen Techniken besteht, so zum Beispiel FPGAs, die integrierte Routing Layer

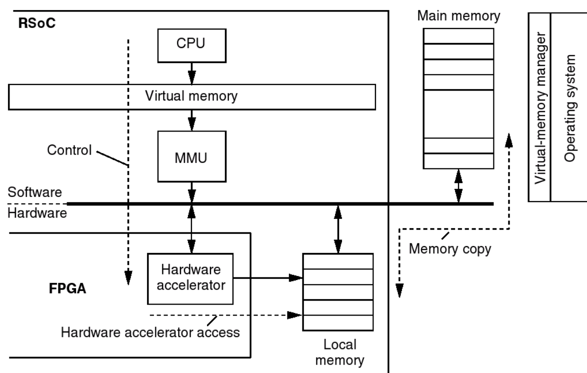

Abbildung 6: Hardware- Speicherzugriff [MV]

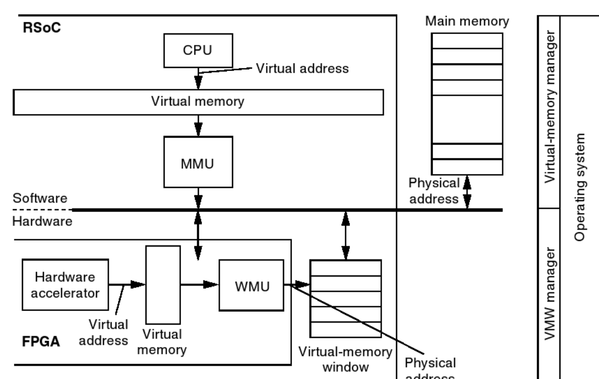

Abbildung 7: Hardware- Speicherzugriff mit virtuellem Speicher [MV]

bieten.

2.6 Speicherverwaltung

Eine sehr wichtige Rolle spielt in jedem Betriebssystem die Speicherverwaltung. Dabei geht es nicht nur um Sicherheitsaspekte, welche vor allem durch Hardware-Erweiterungen hinzugekommen sind (moderne MMUs, Memory Management Units), sondern eher um das Erzeugen eines virtuellen Adressraums. Mit der Einführung des virtuellen Adressraums konnten erstmals Prozesse erzeugt werden, die beliebig oft auf dem selben Rechner parallel laufen können, ohne sich gegenseitig den Speicher zu überschreiben, da nun keine absoluten realen Adressen benutzt werden, sondern virtuelle, die das Betriebssystem in reale Adressen erst umrechnet.

So sieht man in Abbildung 6 eine bisher verwendete Struktur in eingebetteten System und SoC (System on Chips). Dabei verwendet die Software einen virtualisierten Speicher und die Hardware greift direkt auf reale Adressen zu. Benötigt die Hardware nun neue Daten, so kopiert die Software, auch mit DMAC- (Direct Memory Access Controller) Unterstützung, neue Daten in den Puffer der Hardware. Das ist in herkömmlichen SoCs auch kein Problem, wenn man jetzt aber dynamisch rekonfigurierbare Einheiten hinzunimmt, die selbstständig arbeiten sollen, dann treten dieselben Probleme zu Tage, die man auch früher mit den Programmen hatte. Man kann nun nicht mehr denselben Task (Hardware- Prozess) mehrmals allozieren, da sie denselben Speicherbereich nutzen würden. Außerdem würde der Speicherbereich eines gerade nicht geladenen Tasks nicht anderweitig genutzt werden können.

Um diesen Missstand zu beseitigen, führt man auch in der Hardware ein Gegenstück zur MMU des Prozessors ein, hier Window Management Unit, kurz WMU, genannt. Dies folgt aus der Überlegung, dass eine Hardware meist mit größeren Datenblöcken arbeitet und man so auch einfach nur virtuelle Blöcke (Windows) bereitstellen muss. Als ein weiterer Vorteil ergibt sich dadurch, dass neben dem Betriebssystem keine, die Hardware kontrollierende, Software vorhanden sein muss. Als Ergebnis dieser Einführung erhält man ein System, in dem die Hardware autonom auf dem Hauptspeicher arbeiten kann und dieselben Funktionseinheiten mehrmals vorhanden sein können. Des Weiteren werden auch keine Speicherbereiche ungenutzt blockiert, das Betriebssystem erhält die vollständige Kontrolle zurück.

Einige der oben besprochenen Punkte wurden in der Realität kaum implementiert, da die aktuelle Technik der FPGAs diese nicht begünstigen. So kann man zur Zeit zum Beispiel FPGAs der Firma Xilinx lediglich spaltenweise rekonfigurieren, wodurch das Platzieren auf die Auswahl eines verfügbaren Slots reduziert und damit quasi eliminiert wird. Durch das Fehlen von gut integrierbaren Schnittstellen zur Manipulation der Netzlisten von Tasks (ähnlich Xilinx JBits) muss auch das Online-Routing entfallen. Die Generierung von Tasks für die Spaltenslots wird zur Zeit gut vom Xilinx Design Flow durch das Modular Design Package unterstützt, daraus ergibt sich die Benutzung des Busses als bevorzugte Topologie. [Xil04]

3 Beispiele

Wie die Umsetzung der oben vorgestellten Teile eines Hardware- Betriebssystems erfolgt(e) und zu welch interessanten Systemkonfigurationen und -ansichten dies führt, wird in diesem Abschnitt gezeigt. Die folgende Liste stellt einen kleinen Überblick über einige Projekte im Bereich Hardware- Betriebssysteme dar. Es gibt mittlerweile auch kommerzielle Projekte, zum Beispiel im Bereich Wearable Computing und natürlich seit einiger Zeit schon im militärischen Sektor. Auf die mit einem Pfeil gekennzeichneten Beispiele wird unten näher eingegangen.

⇒ SCORE (Stream Computations Organized for Reconfigurable Execution), U Berkeley

⇒ Prototype OS, ETH Zürich

⇒ HTHREAD (hybrid thread), U Kansas

- ARCS (ein NoC), U Rostock
- DISC (Dynamic instruction set computer), U Provo
- DynaCore, U Lübeck
- OS4RS (Operating System for Reconfigurable Systems), U Brüssel / U Leuven
- HOPES (Hardware Operating System), U Singapore
- ReConfigME, U Berkeley

3.1 SCORE

SCORE verfolgt einen anderen Systemansatz als fast alle anderen gesichteten Projekte und daher möchte ich zuerst darauf eingehen. Sie verwenden logisch eine gepipelinete Topologie, die sie physikalisch auf ein Netzwerk abbilden. Das Vorgehen ist in Abbildung 8 genauer zu erkennen. Als erster der verschiedenen Funktionsblöcke sticht einem die CPU (uP) mit ihren Caches (L1, i$, L1 d$, L2 $) ins Auge, auf der das Betriebssystem ausgeführt wird. Als nächstes erkennt man die Compute Pages (CP) als dynamisch rekonfigurierbaren Anteil des Systems und die konfigurierbaren Speicherblöcke (Configurable Memory Block, kurz CP). Die Compute Pages enthalten als so genannte Operatoren die eigentlichen Funktionsblöcke dieses datenstromorientierten Systems. Ein Vorteil der gewählten Vorgehensweise ist im virtualisierten Speicher begründet, so kann

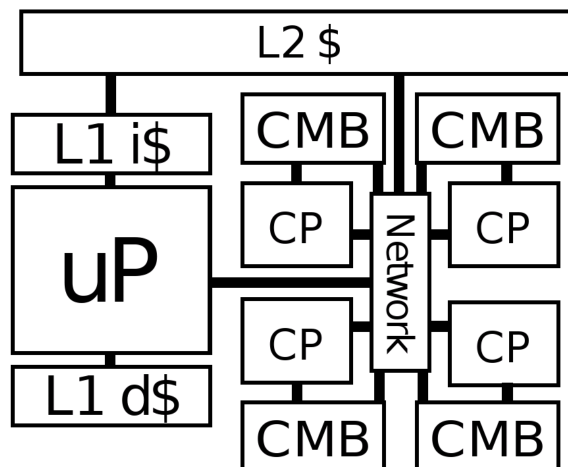

Abbildung 8: SCORE: hypothetischer single chip Entwurf [EC05]

ein CMB als Puffer konfiguriert werden, aus dem gelesen und in den geschrieben werden kann. Diese Puffer können vom Betriebssystem online ein und ausgelagert werden, womit sich für die CPs ein quasi unbegrenzt großer Speicher ergibt. Weiterhin enthalten diese Speicherblöcke auch noch Programmvariablen und werden mit dem zugehörigen Task oder Compute Page (de-) alloziiert werden, damit sein Status erhalten bleibt.

Alle Anwendungen sind im System auch noch als formale Modellierung bezüglich des Datenflusses vorhanden, damit die Operatoren der Anwendung zur richtigen Zeit in eine Compute Page geladen werden können. Außerdem werden daraus auch die Verbindungen über das Netzwerk bestimmt. Reicht die Anzahl der zur Verfügung stehenden Compute Pages nicht aus, so werden die entsprechenden Streams in Puffer geschrieben und später von einem anderen Operator weiterverarbeitet. Generell werden alle Daten von Speichern über Operatoren zu Speichern bewegt. So kann man in Abbildung 9 sehen, wie die Datenströme über Speicher (hier Segment) von oben und unten über Operatoren (hier Virtual Compute Page, VCP) nach rechts fließen. Von links kommt ein Datenstrom aus einem nicht dargestellten CMP/CP und rechts geht es ebenso weiter. Man kann sich dann auch spezielle Ein- und Ausgabeblöcke vorstellen, die als Treiber in der Regel eher fest alloziiert sein werden.

Insgesamt stellt die Gruppe um DeHon und Wawrzynek an der Universität Berkeley mit SCORE ein sehr

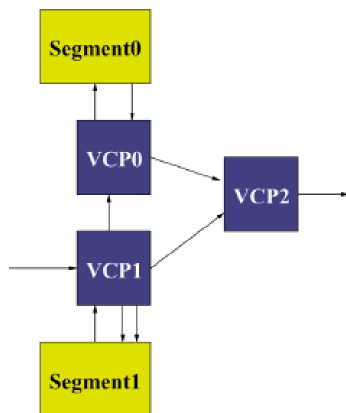

Abbildung 9: SCORE: Programmiermodell [EC05]

gut skalierendes System vor, welches flexibel und leistungsfähig ist. Durch ihr datenstromorientiertes Programmiermodell wird die Hardware gut abstrahiert und man könnte sich auch andere Plattformen als Teil eines solchen Systems vorstellen. Es erscheint durchaus denkbar, dass dieses System ohne große Änderungen auf ein Cluster übertragbar wäre.

3.2 Prototype OS

Das Projekt an der ETH Zürich, hier Prototype OS genannt, verfolgt eine andere Systematik. Es setzt auf ein Taskmodell ähnlich dem Software Thread Modell auf. In Abbildung 10 erkennt man die dazu nötigen Teile des Betriebssystems. Zuerst einmal gibt es einen Task Scheduler, der ähnlich einem herkömmlichen Scheduler für Software, Tasks mit Hilfe der Task Preparation Unit instantiieren kann und damit über den Resource Manager den Placer anstößt. Es gibt weiterhin einen Zeitmanager, für die Systemzeit und andere Timer. Die Task Preparation Unit erzeugt nun bei Instantiierung eines Tasks von dem Rohtask aus dem Raw Task Repository und dem Task- Kontext aus dem Task Context Store einen lauffähigen Task, der sich in dem Zustand befindet, in dem er unterbrochen und aus dem FPGA zurückgelesen wurde. Der schon erwähnte Resource Manager verwaltet beispielsweise die den Tasks zur Verfügung stehenden Gerätetreiber im FPGA und Warteschlangen.

Über den FPGA-Treiber und den Configuration and Readback Port des FPGAs sowie ein Bus Interface, mit GPIOs (General Purpose Input Output) realisiert, ist für eine Anbindung der CPU an das FPGA gesorgt. Im FPGA gibt es noch eine Memory Management Unit

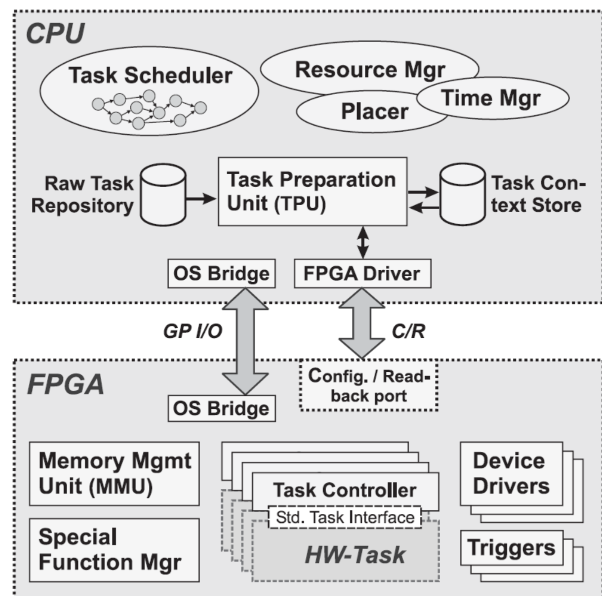

Abbildung 10: Prototype OS: Betriebssystemkomponenten [WP03]

für die Hardware Tasks, über die dann der FPGA interne Speicher und externe Speicher verfügbar sind. Daneben gibt es noch Trigger als Eventquellen (z.B. von Schaltern) und einen Special Function Manager, der die im FPGA verfügbaren Spezialfunktionen wie Multiplizierer oder DSP- (Digital Signal Processing) Einheiten verwaltet.

In diesem System werden Tasks durch ihre geometrische Größe in Form eines Rechtecks und der Taktbandbreite definiert, damit das Betriebssystem entsprechende Slots finden und die Taktfrequenz für das Modul passend bereitstellen kann. Tasks sind durch die oben erwähnten technischen Einschränkungen, wie in Abbildung 11 ersichtlich, als Spalten im FPGA realisiert, welche zwischen betriebssystemeigenen Spalten liegen. Letztere dienen vor allem der Kommunikation und umfassen die Speicherblöcke, Spezialfunktionsblöcke sowie die I/O Blöcke im FPGA.

Die Kommunikation läuft vom Task aus genauer betrachtet über das Standard Task Interface (STI) ab, über welches er mit dem Betriebssystem kommunizierend mit anderen Tasks Informationen austauschen kann oder Betriebssystemdienste anfordert. Die Inter-Frame Communication Channels (IFCC) müssen von jedem Task bereitgestellt werden, damit die zwischen den Tasks liegenden Betriebssystemframes erreichbar

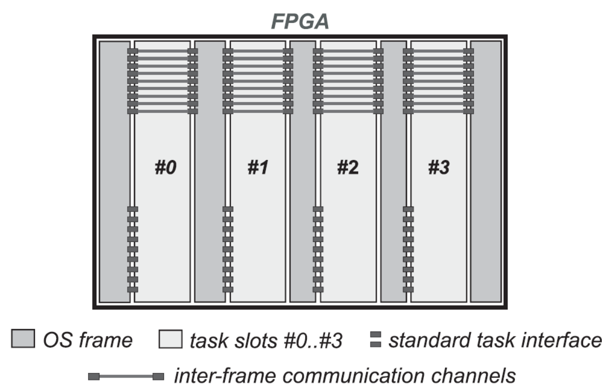

Abbildung 11: Prototype OS: Tasks und OS im FPGA [WP03]

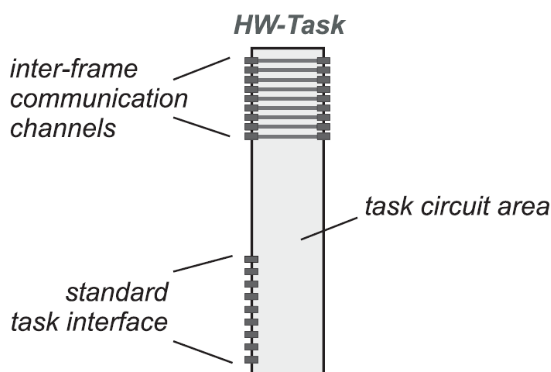

Abbildung 12: Prototype OS: Hardware- Taskinterfaces [WP03]

sind. Daher wird, wenn kein spezieller Task geladen wird, ein Dummy-Task geladen, der die IFCC implementiert. In Abbildung 14 ist so ein Task mit IFCC und STI dargestellt. Die IFCC, als Pflichtbestandteil eines jeden Tasks, sind natürlich nicht nutzbare Flächen und ergeben somit einen systembedingten Overhead. Die Tasks dieses Systems sind immer gleich groß und werden mit Hilfe von Bus Makros und des Modular Design Packages (mittlerweile im ISE Paket integriert) von Xilinx [Xil04] erzeugt.

An diesem Projekt von Walder und Platzner hat man nun schon gesehen, dass Hardware Tasks den Software Tasks ähnlicher werden. Im nächsten Beispiel wird das nun auf die Spitze getrieben.

3.3 Hybrid Thread

Das HTREAD Modell geht davon aus, dass Software und Hardware eine einheitliche Schnittstelle bekommen sollen, damit der Programmierer oder Hardware Ersteller nicht immer umdenken oder sich Gedanken machen muss, ob die Funktion, die man benutzt, in Hardware oder Software abläuft. Dieses sehr elegante Modell geht von POSIX (Portable Operating System Interface) Threads aus und gliedert die Hardware dort ein. Als Programmiersprachen kamen hier VHDL, C und GIMPLE zu Anwendung. Wobei gerade eine eigene Beschreibungssprache für einen homogenen Software- und Hardware- Designflow entwickelt wird, in der dann alle Threads beschrieben werden können und nach Wahl bei Generation des Systems in Hard- oder Software verlagert werden. So wurde sogar das Betriebssystem zum Großteil in Hardware realisiert. Dabei hängen alle Bestandteile des Systems, siehe Abbildung 13 an einem Bus, über den alle typischen Betriebssystemdienste wie Synchronisationsdienste (Mutexe, Zustandsvariablen) bereitgestellt werden. Die Thread Scheduler- und Manager werden folglich nur noch als Gerät betrachtet, die die Threads (Hardware- oder Software-) kontrollieren und verwalten.

Der CBIS (CPU Bypass Interrupt Scheduler) enstammt einem guten, aber ungewohnten Konzept der Interruptverarbeitung, welches auf den besonderen Gegebenheiten des Gesamtsystems beruht. Es werden wie in bekannten Interruptcontrollern Interrupts maskiert und erfasst, aber letztendlich wird nicht die CPU unterbrochen, sondern der CBIS kommuniziert mit dem Scheduler und sorgt damit eventuell für einen Threadwechsel und damit zu einer Interruptbearbeitung. So wird das Problem der Zuweisung von Interruptprioritäten einfach auf die Threadpriorität der Interrupt Service Routine (ISR) verlagert. Jeder Thread kann bei dem CBIS Interrupts an sich binden und so zur ISR werden. Konflikte zwischen verschiedenen Interrupt Service Routinen werden nun über den Scheduler und die Threadprioritäten gelöst. Da (zur Zeit zwar noch nicht, aber in Zukunft) auch Hardware- Threads dynamisch gestartet werden können, kann eine Interruptbehandlung sogar gänzlich an der CPU vorbei erfolgen.

Um Hardware- Elemente so nahtlos in eine ursprüngliche Softwareumgebung einzupassen und für den Programmierer gleich darzustellen ist eine Abstraktionsebene nötig, die auch hier Hardware- Taskinterface heißt. Dieses Interface beinhaltet einige Register und kleine State Machines, die zum Einen die Buszugriffe

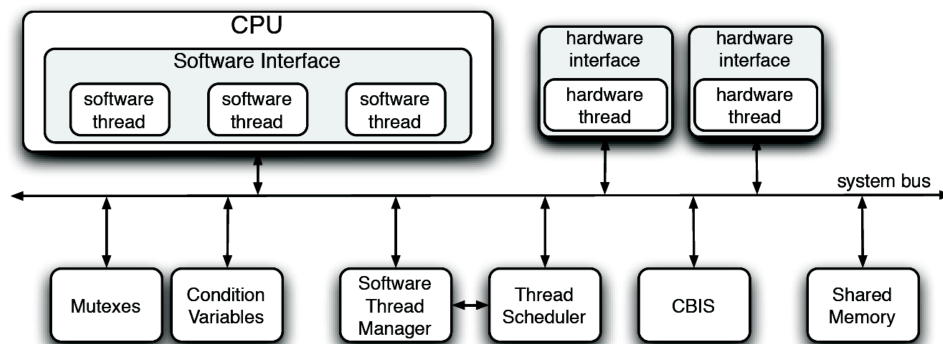

Abbildung 13: HTHREADS: Systemübersicht [DA05]

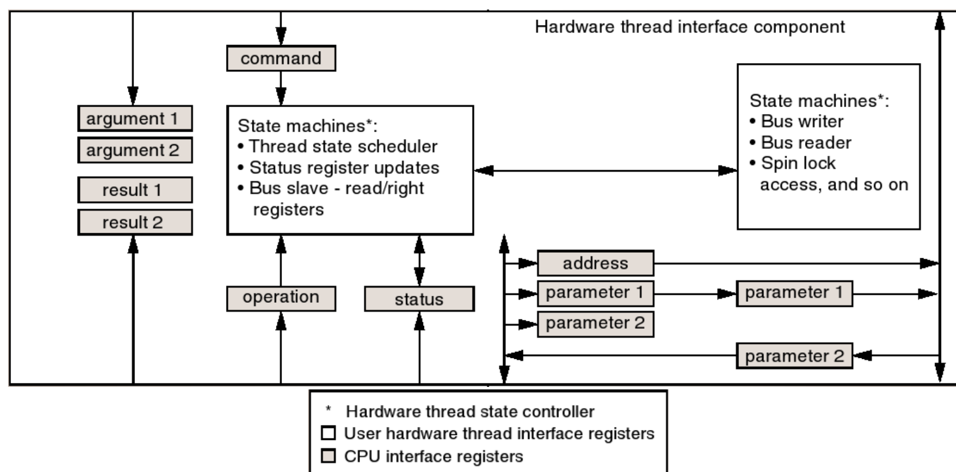

Abbildung 14: HTHREADS: Hardware- Taskinterface [DA04]

kontrollieren und zum Anderen den Prozesszustand verwalten. So wird bei einem Lock der Hardware einfach der Takt entzogen und später wieder zur Verfügung gestellt. Die Register des Task Interfaces stellen zum Teil einen kleinen Stack mit zwei Argumenten und Ergebnissen dar, welche natürlich auch Pointer sein können, um komplexe Datenstrukturen handhaben zu können. Mit diesem Interface wurde nun threadseitig der Bus abstrahiert, man braucht nur noch Registerzugriffe, und betriebssystemseitig wurde die Hardware abstrahiert.
Für eingebettete Systeme bietet dieser Ansatz einen großen Vorteil, so fällt bei einem Blick auf Abbildung 15 auf, dass die Zeitdifferenzen zwischen den Messungen mit 2 Threads und 250 Threads relativ gering sind. Dieser geringe Jitter ist bedingt durch die parallele Verarbeitung und Verwaltung innerhalb des Betriebssystems mit Hardware- Unterstützung, wodurch dann im entscheidenden Augenblick (zum Beispiel beim Prozesswechsel) nur noch ein linearer oder konstanter Aufwand zu bewältigen ist.

Mit diesem, sich noch in der Entwicklung befindlichen System, ist der Gruppe um Andrews, an der Universität Kansas, eine durchaus gelungene Implementation eines vielleicht wegweisenden Konzepts gelungen.

4 Zusammenfassung

Nach Entwicklung und Adaption von Algorithmen aus anderen Bereichen, wurden die ersten Betriebssysteme implementiert. Die ersten Demonstratoren wurden in Betrieb genommen und es zeigte sich, dass es nicht nur möglich ist mit solchen Systemen zu arbeiten, sondern, dass sie ihre Vorteile auch ausspielen können.

	2 Running Software Threads			250 Running Software Threads		
	Min (μs)	Mean (μs)	Max (μs)	Min (μs)	Mean (μs)	Max (μs)
Scheduling Decision	1.750	1.751	2.140	1.910	1.975	3.380
Mutex Lock	.750	.750	.750	.750	.750	.750
Interrupt Handler Determination	.760	.760	.760	.760	.796	1.530

Abbildung 15: Performance vom HTHREAD Betriebssystem [EA]

Letztendlich kommen homogenere Designansätze auf, es wird nach High Level Languages (HLL) gesucht und schließlich in Eigenregie implementiert. Der Trend zur größeren Abstraktion, nicht nur durch HLLs setzt sich fort, so werden auch formale Beschreibungen wieder aufgegriffen.

Es zeigt sich, dass trotz der zur Zeit manchmal einengenden technischen Möglichkeiten bei der partiellen Rekonfiguration der FPGAs oder des großen Kommunikationsoverheads bei NoCs, Systeme entstehen, die leistungsfähig und effizient arbeiten, wenn man die Einschränkungen als solche erst einmal akzeptiert.

5 Ausblick

Gerade auf der Ebene der höheren Abstraktion und allgemeinen Beschreibungssprachen, welche dringend für einen neuen Entwurfsfluss im Hardware- Software Co-Design, benötigt werden, zeigt sich erhebliches Entwicklungspotenzial. Diese spielen insbesondere im Hinblick auf hybride Betriebssysteme und deren homogenen Ansätze eine besondere Rolle.

Literatur

[DA04] Razali Jidin Michael Finley Wesley Peck Michael Frisbie Jorge Ortiz Ed Komp Peter Ashenden David Andrews, Douglas Niehaus. Progamming models for hybrid fpga-cpu computational compnents: A missing link. strony 42–53, 2004. DOI: `http://doi.ieeecomputersociety.org/`.

[DA05] Jason Agron Keith Preston Ed Komp Mike Finleyy Ron Sass David Andrews, Wesley Peck. hthreads: A hardware/software codesigned multithreaded rtos kernel. 2005.

[EA] Wesley Peck Jim Stevens Fabrice Baijot Ed Komp Ron Sass David Andrews Erik Anderson, Jason Agron. Enabling a uniform programming model across the software/hardware boundary.

[EC05] Randy Huang Joseph Yeh Yury Markovskiy André DeHon John Wawrzynek Eylon Caspi, Michael Chu. Stream computations organized for reconfigurable execution (score): Introduction and tutorial. 2005.

[MV] Paolo Ienne Miljan Vuletic, Laura Pozzi. Seamless hardware-software integration in reconfigurable computing systems.

[Wig05] Grant Brian Wigley. *An Operating System for Reconfigurable Computing*. Praca doktorska, University of South Australia, 2005.

[WP03] H. Walder, M. Platzner. Reconfigurable hardware os prototype, 2003.

[Xil04] Xilinx. Two flows for partial reconfiguration: Module based or difference based. Raport instytutowy, Xilinx, 2004.