MARCO PLATZNER, CHRISTIAN PLESSL

VERSCHIEBUNGEN AN DER GRENZE ZWISCHEN HARDWARE UND SOFTWARE

1. Einleitung

Im Bereich der Computersysteme ist die Entscheidung, welche Funktionen in Hardware und welche in Software erledigt werden, eine zentrale Problemstellung. Die Festlegung dieser Grenze hat in den letzten Jahrzehnten nicht nur die Entwicklung von Computersystemen bestimmt, sondern auch die Strukturierung der Ausbildung in den Computerwissenschaften beeinflusst und sogar zur Entstehung von neuen Forschungsrichtungen geführt.

In diesem Beitrag beschäftigen wir uns mit Verschiebungen an der Grenze zwischen Hardware und Software und diskutieren insgesamt drei qualitativ unterschiedliche Formen solcher Verschiebungen. Wir beginnen mit der Entwicklung von Computersystemen im letzten Jahrhundert und der historischen Entstehung dieser Grenze, die Hardware und Software erst als eigenständige Produkte differenziert. Dann widmen wir uns der Frage, welche Funktionen in einem Computersystem besser in Hardware und welche besser in Software realisiert werden sollten. Da sich viele Funktionen funktional äquivalent in Hardware oder Software realisieren lassen, hat die systematische Untersuchung und Entscheidung dieser Fragestellung zu Beginn der 90er Jahre zur Bildung einer eigenen Forschungsrichtung, dem sogenannten Hardware/Software-Codesign, geführt. Im Hardware/Software-Codesign findet eine Verschiebung von Funktionen an der Grenze zwischen Hardware und Software während der Entwicklung eines Produktes statt, um Produkteigenschaften zu optimieren. Im fertig entwickelten und eingesetzten Produkt hingegen können wir dann eine feste Grenze zwischen Hardware und Software beobachten. Im dritten Teil dieses Beitrags stellen wir mit selbst-adaptiven Systemen eine hochaktuelle Forschungsrichtung vor. In unserem Kontext bedeutet Selbst-Adaption, dass ein System Verschiebungen von Funktionen an der Grenze zwischen Hardware und Software autonom während der Betriebszeit vornimmt. Solche Systeme beruhen auf rekonfigurierbarer Hardware, einer relativ neuen Technologie, mit der die Hardware eines Computers während der Laufzeit verändert werden kann. Diese Technologie führt zu einer durchlässigen Grenze zwischen Hardware und Software und löst damit die herkömmliche Vorstellung einer festen Hardware und einer flexiblen Software auf.

2. Von Hardware zu Software

Der Begriff „Hardware“ bezeichnet die physikalischen Teile eines Computers, die Teile, die – wenigstens im Prinzip – anfassbar sind. So steht das Wort „hardware“ im Englischen auch für Eisenwaren und ein „hardware store“ ist ein Eisenwarenladen. Computerhardware umfasst zum Beispiel die Prozessoren, Speicher und externe Datenträger, Gehäuse, Tastaturen und Mäuse. Der Begriff „Software“ bezeichnet die Programme und manchmal auch Daten, also Teile, die nicht anfassbar sind. Software wird üblicherweise klar abgegrenzt von den Rechenvorschriften bzw. Algorithmen, die einen Problemlösungsprozess für einen Computer formal beschreiben. Software bezeichnet dann konkrete Umsetzungen der Algorithmen in Programmiersprachen, wobei es von sehr abstrakten anwendungsorientierten Sprachen bis hin zu Maschinensprachen ein breites Spektrum an Programmiersprachen gibt. Jedenfalls lässt sich feststellen, dass die Ausführung von Software Hardware voraussetzt, und auch für die Speicherung von Software müssen geeignete Datenträger vorhanden sein.

1 – ENIAC, University of Pennsylvania

Die ersten Computer verwendeten noch keine Software im heutigen Sinne. Die verwendeten Programme wurden nicht explizit auf einem Datenträger gespeichert, sondern durch Verkabelung festgelegt. Abbildung 1 zeigt einen der ersten Computer, den ENIAC (Electronic Numerical Integrator and Calculator), der von J. P. Eckert und J. Mauchly an der University of Pennsylvania entwickelt und im Jahr 1946 der Öffentlichkeit vorgestellt wurde. ENIAC war zwar ein programmierbarer General-Purpose-Computer, der grundsätzlich durch seine Programmierbarkeit für unterschiedliche Aufgaben eingesetzt werden konnte, wurde aber hauptsächlich für militärische Aufgaben wie bal-

listische Berechnungen eingesetzt. Als einer der ersten elektronischen Computer war ENIAC 80 Fuß lang und mehrere Fuß hoch. Die Programmierung erfolgte durch Verkabelung, die Eingabe der Daten über damals übliche Lochkarten.

Die Idee, Programme gleich wie Daten zu behandeln, wird heute meistens John von Neumann zugeschrieben[1] und bedeutete damals, Programme über Lochkarten in einen Computer einzulesen und dort im Speicher abzulegen. Software explizit zu speichern (Stored Program Concept) hatte technische und wirtschaftliche Vorteile und die Idee fand rasch Verbreitung, zum Beispiel im EDSAC (Electronic Delay Storage Automatic Calculator) 1949 in Cambridge, UK. Frühen Computern wie ENIAC und EDSAC, die entweder für militärische Anwendungen entwickelt wurden oder Forschungsprojekte waren, folgten rasch kommerzielle Computerentwicklungen. Beispiele sind der UNIVAC im Jahr 1951, der 48 Mal verkauft wurde und für die Vorhersage der Ergebnisse der Präsidentenwahlen in den USA Bekanntheit erlangte, oder das IBM System 701 im Jahr 1952, das insgesamt 19 Mal verkauft wurde und den relativ späten Einstieg von IBM in das Geschäft mit Computern markiert.

In den Anfängen der Computertechnik waren Hardware und Software keine getrennten Komponenten, sondern integrale Bestandteile eines Computers. Computer wurden von einer Firma entworfen, hergestellt und als Gesamtsystem verkauft. Die Software wurde oft auch noch eingeteilt in BIOS (Basic Input/Output System) für grundlegende Funktionen zu Ein- und Ausgaben von Daten, Betriebssystem, Systemsoftware (Lader, Assembler, Compiler, Linker) und Anwendersoftware. Die Anwendersoftware wurde entweder auch vom Computerhersteller mitgeliefert oder, was zunehmend der Fall war, vom Kunden angepasst oder erstellt. Software wurde erst nach und nach zu einem eigenständigen Produkt, getrieben durch die rasante Verbreitung von Computern und unterstützt durch die Etablierung von quasi-standardisierten Computersystemen.

Im Jahr 1964 brachte IBM das System/360 auf den Markt, das erstmals den Ansatz der Computerfamilie umsetzte. Mit dem System/360 konnte IBM sechs Varianten eines Computers mit unterschiedlichem Preis-Leistungs-Verhältnis anbieten. Die Idee der Computerfamilie war äußerst erfolgreich, da Investitionen in Software auf unterschiedlich leistungsfähigen Computern genutzt werden und damit den Kundenbedürfnissen angepasst werden konnten. Das System/360 und seine Nachfolger dominierten bald den Computermarkt. Unterstützt durch den Fortschritt in der Mikroelektronik brachte DEC im Jahr 1965 mit der PDP8 einen sogenannten Minicomputer am unteren Ende der Preisskala für weniger als 20.000 US-Dollar auf den Markt. Diese neue Klasse von kleineren, günstigeren und dennoch leistungsfähigen Computersystemen trug stark zur weiteren Verbreitung von Computern bei und erlaubte viele neue Anwendungen, zum Beispiel in der Steuerung von Produktionsanlagen.

[1] Vgl. Paul E. Ceruzzi, *A History of Modern Computing*, 2. Aufl., Cambridge, MA, 2003.

Die PDP8 wird häufig als Vorläufer des Mikroprozessors gesehen, eines Prozessors, der aus nur einem einzigen Chip besteht. Der erste Mikroprozessor war dann der Intel 4004, der 1971 eingeführt wurde. In den folgenden Jahrzehnten hielt der Mikroprozessor Einzug in alle Klassen von Computersystemen, von kleinen eingebetteten Computern über den Personal Computer bis hin zu Supercomputern.

General-Purpose-Computer sind per Konstruktion für viele Anwendungen geeignet. Eine Konsequenz aus dem General-Purpose-Konzept ist, dass man bei der Herstellung der Hardware die Software (noch) nicht kennen muss. Diese Unabhängigkeit, die weite Verbreitung und die Quasi-Standardisierung von Computerhardware erlaubte es, zunehmend Software als eigenes Produkt zu etablieren. In den 70er Jahren wurden viele Firmen gegründet, die ausschließlich Software entwickelten, zum Beispiel Microsoft und SAP. Diese Öffnung des Computermarktes und die Loslösung der Software von der Hardware im Entwurfs- und Herstellungsprozess kann man als Entstehung der Grenze zwischen Hardware und Software sehen. Bei General-Purpose-Systemen hat der technische Fortschritt im Hardwarebereich zu einer klassischen „Commoditization" geführt, wie sie in der Wirtschaftsliteratur z. B. von Christensen[2] beschrieben wird. Das heißt, General-Purpose-Hardware ist weitgehend standardisiert und die Produkte der einzelnen Hardwarehersteller sind nicht mehr wesentlich voneinander differenziert. Dies hat zu einer Modularisierung der Komponenten und Geschäftsmodelle sowie einer starken Erosion der Gewinnmargen der Hardwarehersteller geführt, so dass heute der Großteil der Wertschöpfung im Bereich der Softwareanbieter liegt.

Neben General-Purpose-Computing gibt es allerdings auch grundlegend andere Domänen mit anderen Geschäftsmodellen. Hierbei sind aktuell besonders die Domäne der mobilen Personal Digital Assistants (PDA), d. h. Mobiltelefone und Tablets, sowie der Bereich des Cloud-Computing hervorhebenswert. Im PDA-Bereich ist momentan noch kein Trend zur Modularisierung zu erkennen. Im Gegenteil, angetrieben durch den Erfolg des integrierten Geschäftsmodells von Apple, ist auch beim Rest dieser Industrie ein deutlicher Trend zu integrierten Produkten und Geschäftsmodellen zu sehen, wie die Übernahme der Smartphone-Sparte von Nokia durch Microsoft oder die Übernahme der Motorola Smartphone-Sparte durch Google zeigen. Nach Christensen ist diese Zuwendung zu einem integrierten Geschäftsmodell ein Indikator dafür, dass die Hard- und Software für mobile Geräte die Kundenbedürfnisse noch nicht hinreichend befriedigt und sich Innovationen nur durch eine passgenaue Abstimmung von Hardware, Betriebssystem und Anwendungssoftware sowie durch eine größere Kontrolle des Herstellers über diese Systeme erreichen lassen. Im Bereich des Cloud-Computing ist interessanterweise ein entgegengesetzter Trend zu verzeichnen. Eine Grundidee des Cloud-Computing ist es gerade, die Eigenschaften der Hardware möglichst zu verbergen und gewissermaßen eine perfekte Com-

[2] Clayton M. Christensen, *The Innovator's Dilemma*, Boston, MA, 1997.

moditization von Rechenleistung und Datenhaltung zu schaffen. Das heißt, die Entscheidung, welche Software wann auf welcher konkreten Hardware ausgeführt wird, ist für den Benutzer nicht nachvollziehbar, was durch eine Abstraktion von Software in sogenannte Dienste erreicht wird. Nicht zuletzt durch den großen Kapitalbedarf für den Aufbau und den Betrieb von Cloud-Systemen ist der Großteil der Kapazität gegenwärtiger Cloud-Systeme heute allerdings in den Händen weniger Anbieter konzentriert (z. B. Google, Amazon, Microsoft, Salesforce). Diese Unternehmen bieten neben der reinen Bereitstellung der Hardware- und Betriebssystem-Infrastruktur zunehmend auch komplexere Plattformen (z. B. Google AppEngine, Amazon Elastic Beanstalk) und auch Anwendungen an (GoogleDocs, Microsoft Office 365), was langfristig auch auf ein integriertes Modell hinauslaufen könnte.

Wie sich die Märkte für Hardware und Software für Computersysteme, getrieben durch technologische Innovationen und neue Anwendungen, weiterentwickeln und ob sich auf diesen Märkten langfristig integrierte oder modulare Geschäftsmodelle durchsetzen werden, ist noch offen. Eine mögliche Zukunft solcher Märkte und der dazugehörigen technischen Umgebungen wird an der Universität Paderborn im Sonderforschungsbereich 901, *On-The-Fly Computing*[3], von einem Forscherteam aus Informatikern und Wirtschaftswissenschaftlern untersucht.

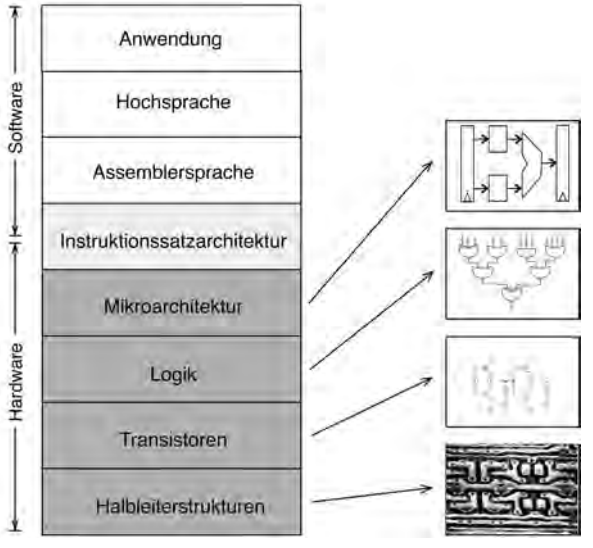
3. Hardware/Software-Co-Design

In diesem Abschnitt besprechen wir, wie sich Computersysteme durch ein Schichtenmodell von aufeinander aufbauenden Hard- und Softwareschichten beschreiben lassen. Diese Betrachtung führt uns zum Konzept der Instruktionssatzarchitektur, welche als Schnittstelle zwischen Hardware und Software dient. Schließlich stellen wir das Forschungsgebiet des Hardware/Software-Codesigns vor, das sich mit der systematischen Optimierung der Grenze zwischen Hard- und Software zum Entwurfszeitpunkt beschäftigt.

3.1 Schichtenmodell für Computersysteme

Der Aufbau von Computersystemen und deren Strukturierung in Hard- und Software wird üblicherweise durch ein Schichtenmodell beschrieben. Jede Schicht stellt dabei eine Abstraktionsebene dar, welche es den höher liegenden Schichten erlaubt, die Funktionen der darunter liegenden Schichten über eine definierte Schnittstelle zu nutzen, ohne deren detaillierte Eigenschaften zu kennen. Abbildung 2 zeigt eine in der Lehre üblicherweise verwendete Darstellungsform dieses Schichtenmodells.

[3] *On-The-Fly Computing*, online unter: http://sfb901.uni-paderborn.de, zuletzt aufgerufen am 24.04.2014.



2 – Klassisches Schichtenmodell von Computersystemen mit verschiedenen Abstraktionsebenen der Hardware (grau) und Software (weiß)

Die unterste Hardwareebene ist die der Halbleiterstruktur, welche die geometrischen Abmessungen und die Halbleitermaterialien beschreibt, die einen Prozessor als integrierte Schaltung (Chip) umsetzen. Auf der nächsthöheren Stufe werden diese Strukturen als Transistoren abstrahiert, welche die Grundelemente elektronischer Schaltungen bilden. Da Prozessoren digitale Schaltungen sind, wird die Hardware weiter abstrahiert und durch Logikelemente dargestellt, welche die Funktionen von Transistor-Baugruppen durch mathematische Grundoperationen in binärer Logik abbilden. Diese Grundoperationen werden in der sogenannten Mikroarchitektur zu komplexeren Einheiten zusammengefasst, welche die Rechen- und Kontrolleinheiten des Prozessors sowie die vorhandenen Register und Speicher zum Halten von Daten beschreibt. Obwohl diese Schaltungsstruktur die Grundlage für die konkrete Ausführung eines Programms durch den Prozessor darstellt, wird sie gegenüber dem Programmierer völlig verborgen. Stattdessen wird als Schnittstelle zwischen der Hard- und Software die sogenannte Instruktionssatzarchitektur als zusätzliche Abstraktion eingeführt.

Die Instruktionssatzarchitektur beschreibt die Grundoperationen des Prozessors aus Sicht des Programmierers, ohne eine spezifische Hardwareumsetzung vorwegzunehmen. So definiert die Instruktionssatzarchitektur zum Beispiel die Namen und Funktionen von arithmetischen Operationen sowie die Namen der internen Register zum Zwischenspeichern von Daten. Eigenschaften einer konkreten Implementierung, z. B. wie schnell Instruktionen ausgeführt werden, ob die Ausführung parallel oder sequenziell erfolgt oder ob zur

Optimierung der Datenhaltung neben den Registern zusätzliche interne Speicher verwendet werden, bleiben bewusst offen.

Als unterste Softwareebene ist die Assemblersprache eng mit der Instruktionssatzarchitektur verwandt. Bei der Assemblersprache handelt es sich um eine textuelle Repräsentation des Instruktionssatzes ohne weitergehende Abstraktionen. Das Programmieren auf dieser Ebene ist heutzutage lediglich noch für sehr wenige, hardwarenahe Aufgaben gebräuchlich, zum Beispiel für Kernfunktionen des Betriebssystems oder zur Optimierung der Rechenleistung für numerische Berechnungen. In den allermeisten Fällen wird Assemblersprache jedoch nicht direkt vom Programmierer erstellt, sondern aus einem in einer Hochsprache geschriebenen Programm erzeugt. Hochsprachen unterscheiden sich von Assemblersprache dadurch, dass sie abstraktere Programmierkonzepte und eine textuelle Notation anbieten, welche auf eine spezielle Klasse von Anwendungen zugeschnitten sind. So bietet die Programmiersprache Fortran zum Beispiel die Möglichkeit, numerische Berechnungen mit Matrizen, Vektoren und komplexen Zahlen bequem auszudrücken. Ein Hochsprachenprogramm lässt sich dann durch ein Softwareübersetzungswerkzeug (Compiler) automatisch und effizient in Assemblersprache transformieren. Als oberste Softwareschicht kann man die Anwendung betrachten.

3.2 Die Instruktionssatzarchitektur als Grenze zwischen Hardware und Software

Aus der Perspektive der Verschiebung der Grenze zwischen Hardware und Software ist die Instruktionssatzarchitektur von zentraler Bedeutung. Diese Grenze wurde in Industrie und Wissenschaft sehr ausführlich untersucht. Das liegt unter anderem auch daran, dass die Instruktionssatzschicht nicht nur die Grenze zwischen Hard- und Software, sondern auch zwischen wissenschaftlichen Disziplinen markiert. Die Umsetzung eines Instruktionssatzes in Hardware, d. h. der Entwurf einer Mikroarchitektur und deren Implementierung, liegt im Bereich der Elektrotechnik, während die Fragestellungen oberhalb der Instruktionssatzebene Kerngebiete der Informatik sind. Unterschiedliche Implementierungen desselben Instruktionssatzes sind durchaus üblich. In der Tat ist genau diese Trennung zwischen Hard- und Software dafür verantwortlich, dass Computersysteme durch Einsatz von neuen Prozessoren mit verbesserter Mikroarchitektur oder aktuellerer mikroelektronischer Technologie enorme Geschwindigkeitszuwächse erzielen können, ohne dass eine Anpassung der Anwendungssoftware notwendig ist.

Eine Grundfrage bei der Definition eines Instruktionssatzes ist die Festlegung der Grenze zwischen Hard- und Software, d. h. welche Funktionen als Prozessorinstruktionen angeboten werden sollen und welche Funktionen besser als Sequenz von solchen Instruktionen (Programm) umgesetzt werden. Bietet der Instruktionssatz nur einfache Instruktionen, welche wenig Arbeit er-

ledigen (z. B. zwei Zahlen addieren), benötigt man viele Instruktionen, um ein Programm auszuführen. Dadurch wird die Ausführungszeit verlängert, die Hardwarekomplexität des Prozessors bleibt hingegen niedrig. Bietet der Instruktionssatz mächtige Instruktionen, welche viel Arbeit erledigen, wird das Programm kürzer. Die Hardwarekomplexität des Prozessors steigt hingegen, was zu höheren Kosten und niedrigerer Geschwindigkeit führt. Ein klassisches Beispiel für eine komplexe Instruktion ist die Multiplikation. In den frühen Jahren der Computertechnik (1950 bis 1970) konnten Prozessoren aufgrund der hohen Schaltungskomplexität keine Hardwaremultiplizierer integrieren. Stattdessen wurden Multiplikationen, wie bei der schriftlichen Multiplikation, durch Addieren und Schieben in Software ausgeführt. Diese Methode war aber sehr langsam, und mit steigenden technologischen Möglichkeiten wurden ab Mitte der 70er Jahre Multiplizierer in Hardware gebaut. Der Hardwareaufwand dieser frühen Multiplizierer war mit etwa 10.000 Chips exorbitant. Aber durch rasante Fortschritte in der Halbleitertechnologie und -fertigung konnte die Komplexität von Schaltungen, die auf einem Chip realisiert werden können, massiv gesteigert werden. Somit konnte ein Multiplizierer bereits Ende der 70er Jahre auf einem einzigen Chip gefertigt werden und heute multiplizieren nahezu alle Prozessoren in Hardware.

Im Laufe der Zeit hat es immer wieder Experimente mit Instruktionssätzen gegeben, die sehr nahe an oder gar identisch mit Hochsprachen waren. So wurden in den 70er und 80er Jahren spezielle LISP-Processing-Machines entwickelt (z. B. die Texas Instruments Explorer), deren Prozessoren direkt LISP-Code ausführen konnten. LISP ist eine funktionale Programmiersprache, die in den 70er Jahren sehr populär für Problemstellungen aus dem Bereich der künstlichen Intelligenz war. Man hoffte mit solchen Hochsprachen-Prozessoren die sogenannte Semantische Lücke zwischen Programmiersprache und Instruktionssatz zu schließen, und damit auch die Aufgabe der Compiler zu reduzieren oder sie im Extremfall ganz überflüssig zu machen. Diese Ansätze waren allerdings allesamt nicht erfolgreich, da die technische Entwicklung von General-Purpose-Prozessoren zu rasch voranschritt und es aufgrund der beschränkten Größe des LISP-Marktes nicht rentabel war, diese speziellen Prozessoren alle 1,5 bis 2 Jahre in der neuesten Technologie zu realisieren.

Die Frage, ob Prozessoren komplexe (Complex Instruction Set Computing, CISC) oder einfache (Reduced Instruction Set Computing, RISC) Instruktionen nutzen sollen und ob ein Konzept dem anderen grundsätzlich überlegen ist, wird seit Jahrzehnten intensiv debattiert. Eine Übersicht über die wesentlichen Unterschiede von CISC- und RISC-Prozessorarchitekturen ist in Tabelle 1 dargestellt.

Tabelle 1: Vergleich von CISC- und RISC-Prozessoren

CISC **(Complex Instruction Set Computing)**	**RISC** **(Reduced Instruction Set Computing)**
Viele Instruktionen	Wenige Instruktionen
Variable Instruktionslänge	Fixe Instruktionslänge
Viele, komplexe Adressierungsarten	Wenige, einfache Adressierungsarten
Beispiele: Intel IA-32, VAX, IBM/360, Intel 8051	Beispiele: PowerPC, SPARC, MIPS, Alpha, Itanium, AVR

Historisch hatten die meisten Computer komplexe Instruktionen, da frühe Computer primär in Assemblersprache programmiert wurden und durch komplexe Instruktionen kurze und aussagekräftige Assemblerprogramme ermöglicht werden. Die mit komplexen Instruktionen verbundenen Nachteile der hohen Hardwarekomplexität und die damit einhergehende Einschränkung der Ausführungsgeschwindigkeit wurde als weniger gravierend eingeschätzt. Mitte der 80er Jahre wurde diese Frage in zwei Projekten an den amerikanischen Universitäten Stanford[4] und Berkeley[5] systematisch untersucht. Dabei wurde empirisch ermittelt, welche Instruktionen von realen Programmen tatsächlich genutzt werden und welche Speicherzugriffs- und Programmablaufmuster sie dabei verwenden. Das Resultat dieser Untersuchungen war, dass die meisten komplexen Instruktionen kaum benutzt werden und dass man durch eine radikale Vereinfachung des Instruktionssatzes, der Zufügung von schnellem lokalem Zwischenspeicher (Cache) und durch Fließbandverarbeitung von Instruktionen (Pipelining) wesentlich einfachere und leistungsfähigere Prozessoren entwerfen kann. Die aus diesen beiden Forschungsprojekten resultierenden und kommerzialisierten Prozessorarchitekturen SPARC und MIPS läuteten das Zeitalter der RISC-Prozessoren ein und seit den 90er Jahren folgen nahezu alle neu entwickelten Prozessorarchitekturen dem RISC-Konzept. Die große Ausnahme von dieser Regel ist Intels IA-32-Prozessorarchitektur, welche vor der RISC-Revolution entwickelt wurde. Diese Architektur hat aus Kompatibi-

[4] Vgl. John Hennessy/Norman Jouppi/Steven Przybylski et al., „MIPS: A Microprocessor Architecture“, in: *Proceedings of the 15th Workshop on Microprogramming (MICRO)*, Piscataway, NJ, 1982, S. 17-22.

[5] Vgl. David A. Patterson, „Reduced Instruction Set Computers“, in: *Communications of the ACM* 28, 1 (1985), S. 8-21.

litätsgründen bis heute einen CISC-Instruktionssatz, ist intern allerdings ebenfalls als RISC-Architektur organisiert.

3.3 Optimierung der Hardware/Software-Grenze

Die Suche nach der optimalen Grenze zwischen Hard- und Software ist nach wie vor eine relevante Entscheidung beim Entwurf von Computersystemen, welche – getrieben durch neue Anforderungen und Anwendungen – jeweils neu getroffen werden muss. Zum Verständnis der Entwicklung ist es notwendig, zwei grundsätzliche Klassen von Computersystemen zu unterscheiden. Die Klasse der General-Purpose-Computing-Systeme ist nicht auf eine spezielle Anwendung oder Domäne zugeschnitten, sondern versucht für ein sehr breites Spektrum von Anwendungen eine hohe Rechenleistung bei akzeptablem Energieverbrauch abzudecken. Das Anwendungsspektrum reicht hierbei von klassischen Büroanwendungen wie Textverarbeitung, Tabellenkalkulation oder Bildbearbeitung bis zu Serveranwendungen in Rechenzentren, welche Dienste wie Datenbanken, Buchhaltung oder E-Mail anbieten. Computersysteme für diesen Markt sind hochgradig modularisiert und standardisiert, d. h. sie werden durch die Integration austauschbarer Hardwarekomponenten (Prozessoren, Mainboards, Speicherbausteine, Festplatten, Gehäuse, Stromversorgung) aufgebaut und durch ein ebenfalls standardisiertes Betriebssystem verwaltet. Die Software für diese Systeme ist auch modular und wird von einer großen Anzahl von Firmen angeboten. Aufgrund der für die Standardisierung notwendigen Abstimmung von Hardware, Software und Betriebssystemen ist die Innovation an der Hardware/Software-Schnittstelle im General-Purpose-Bereich eher langsam, findet aber dennoch kontinuierlich statt. Zum Beispiel wurde der Intel IA-32-Instruktionssatz in den letzten Jahren mit Befehlen zur Beschleunigung der rechenaufwendigen AES-Verschlüsselungsmethode erweitert, um dem zunehmenden Einsatz dieses Verfahrens Rechnung zu tragen.

Demgegenüber steht die Klasse der Embedded-Computing-Systeme, welche Computer beschreibt, die in technische Systeme integriert sind. Beispiele für solche Systeme sind Autos, Satelliten, Hörgeräte, industrielle Steuerungen, Drucker, Fax, Fernseher etc. Im Gegensatz zu General-Purpose-Systemen üben eingebettete Systeme eine sehr spezifische Funktion aus und sind daher nicht als typische Computersysteme zu erkennen. Sie unterliegen durch die Interaktion mit der physikalischen Welt (z. B. Sensoren, elektro-mechanische Komponenten, Benutzer) auch anderen Entwurfszielen, wie zum Beispiel geringen Kosten, niedrigem Energieverbrauch, hoher Zuverlässigkeit, Sicherheit oder einer ausreichenden Performance für eine klar definierte Aufgabe. Viele dieser Ziele stehen grundsätzlich miteinander in Konflikt und können nicht gleichzeitig maximiert, sondern nur gegeneinander abgewogen werden. Zum Beispiel steht eine hohe Rechenleistung im Konflikt mit geringen Kosten, da zur Erzielung hoher Rechenleistung zusätzliche, spezialisierte Hardwarekomponenten benötigt werden. Folglich müssen Hard- und Software im Bereich

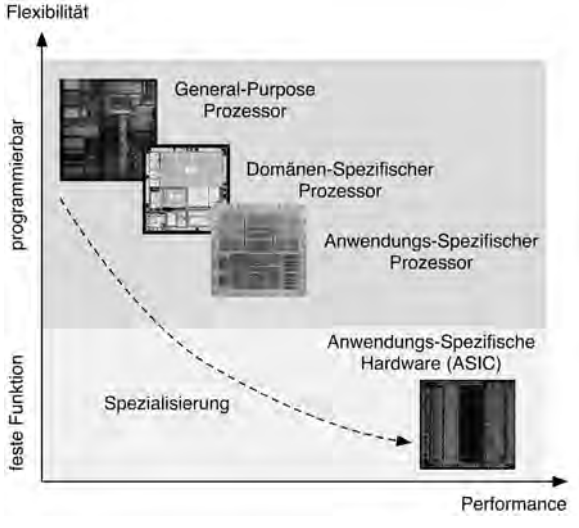
des Embedded Computing sehr genau aufeinander abgestimmt werden, um für das resultierende Gesamtsystem ein optimales Ergebnis zu erzielen. Anbieter in diesem Bereich verfolgen daher statt einem modularen meist ein integriertes Geschäftsmodell, bei dem ein System mit für die Anwendung maßgeschneiderten Hardwarekomponenten und genau darauf abgestimmter Software angeboten wird.

In den 90er Jahren wurde der systematische Entwurf von Computersystemen und deren Optimierung als eigenes, methodisch orientiertes Forschungsgebiet mit dem Namen Hardware/Software-Codesign etabliert.[6] Die Grundidee des Hardware/Software-Codesigns ist dabei eine ganzheitliche Betrachtung des Entwurfsprozesses ausgehend von einer Beschreibung des gewünschten Systemverhaltens. Diese Beschreibung ist unabhängig davon, welche Funktionen letztlich in Software – das heißt als Programm auf einem Prozessor – ausgeführt werden und welche aus Gründen der Performance oder der Energieeffizienz in maßgeschneiderter Hardware umgesetzt werden. Zur systematischen Optimierung werden die Funktionen des Systems modelliert, sowie die gewünschten nichtfunktionalen Eigenschaften und Rahmenbedingungen (z. B. die benötigte Performance, akzeptabler Energieverbrauch, Reaktionszeiten zur Verarbeitung von Sensor und Benutzereingaben) formal erfasst. Dann werden die betrachteten Komponenten (verschiedene Arten von Prozessoren, Hardwarebeschleuniger, Speicher, Kommunikationsnetzwerke) und deren Eigenschaften ebenfalls auf einer abstrakten Ebene modelliert. Durch computergestützte Methoden kann nun eine Auswahl an Ressourcen getroffen und eine optimale Zuordnung von Anwendungsfunktionen zu Soft- und Hardwareressourcen bestimmt werden. Dieser Prozess wird als Hardware/Software-Partitionierung bezeichnet. Im Allgemeinen resultiert dieser Prozess nicht in einer einzigen, optimalen Lösung, sondern in einer Vielzahl von Implementierungsvarianten.

Abbildung 3 illustriert die Ursache eines derartigen Konflikts. Bei der Abbildung von Funktionen auf Rechenressourcen existieren verschiedene Varianten. Eine Variante ist es, die Funktion in Software auf einem programmierbaren General-Purpose-Prozessor auszuführen. Diese Lösung ist sehr flexibel, da grundsätzlich jede Funktion ausgeführt und auch leicht geändert werden kann. Der Allzweckcharakter des Prozessors bedingt allerdings auch Ineffizienzen, welche sich in suboptimaler Performance niederschlagen. Durch die Verwendung eines Prozessors, der für eine spezifische Klasse von Funktionen oder gar für die Anwendung selbst optimiert ist, lässt sich eine bessere Performance erzielen. Diese Lösungen sind allerdings weniger flexibel, da die Umsetzung auf die spezialisierten Rechenressourcen komplizierter ist und sich die Spezialisierung für andere Funktionen negativ auswirken kann. Die optimale Lösung in Bezug auf die Performance ist eine Ausführung in fester, komplett anwen-

[6] Vgl. Jürgen Teich/Christian Haubelt, *Digitale Hardware/Software-Systeme. Synthese und Optimierung*, 2. Aufl., Berlin, Heidelberg, New York, NY, 2007.

dungsspezifischer Hardware. Durch diese Verschiebung von Software zu Hardware entfallen zwar alle Ineffizienzen der Programmierbarkeit, allerdings lässt sich die Funktion im Nachhinein auch nicht mehr anpassen.



3 – Im Hardware/Software-Codesign stehen die verschiedenen Entwurfsziele üblicherweise miteinander in Konflikt. Die Abbildung illustriert einen Trade-Off zwischen den Zielen Flexibilität und Performance. Durch zunehmende Spezialisierung der Rechenressourcen lässt sich eine höhere Performance erzielen, allerdings bewirkt die Spezialisierung eine geringere Flexibilität.

Diese Art der Verschiebung von Funktionen an der Hardware/Software-Grenze ist im Entwurf heutiger Computersysteme, insbesondere für Embedded Systems, essenziell. Dies trifft besonders auf Systeme zu, welche Daten mit sehr hoher Rate verarbeiten müssen oder batteriebetriebene Geräte, welche mit einem sehr knappen Energiebudget haushalten müssen. Im Laufe der Zeit wurden Komponenten, die sich in vielen Systemen als anwendungsspezifische Beschleuniger-Hardware bewährt haben, in Prozessoren integriert, um das Beste aus beiden Welten zu vereinen. So besitzen heute alle domänen-spezifischen Prozessoren für Mobiltelefone spezielle Hardwarebeschleuniger-Einheiten zum Dekomprimieren von Videodaten, ohne die in einem batteriebetriebenen Gerät ein stundenlanges kontinuierliches Abspielen von Videos undenkbar wäre.

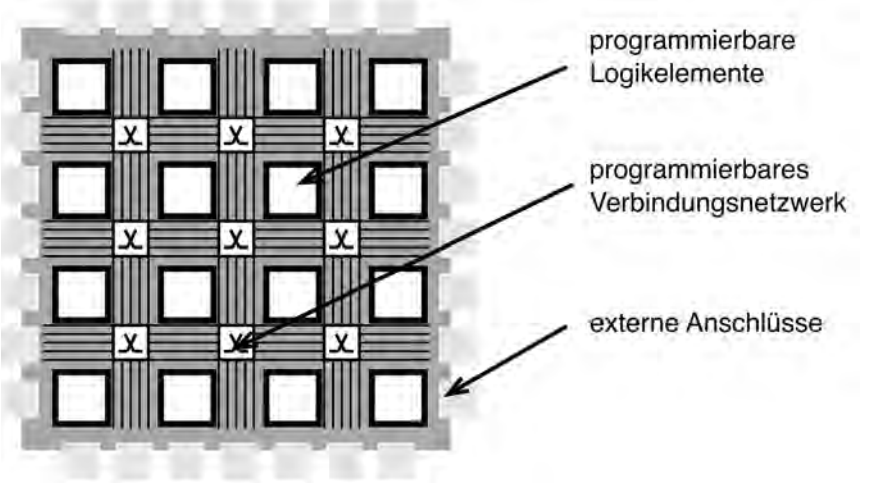
Alle in diesem Abschnitt betrachteten Ansätze verschieben Funktionen an der Grenze zwischen Hardware und Software zur Entwurfszeit. Im nächsten Abschnitt betrachten wir neuartige Systeme, welche diese Grenze während der Laufzeit des Computersystems verschieben, um dynamisch auf veränderte Anforderungen zu reagieren.

4. Hardware/Software-Migration

In diesem Abschnitt beschreiben wir, wie rekonfigurierbare Hardwaretechnologie verwendet werden kann, um Funktionen während des Betriebs eines Computersystems über die Hardware/Software-Grenze hinweg zu verschieben. Diese Hardware/Software-Migration ermöglicht den Aufbau selbst-adaptiver Systeme. Wir umreißen zuerst die Entwicklung der zugrundeliegenden Hardwaretechnologie, beschreiben dann deren Einsatz anhand eines Anwendungsbeispiels auf einem hybriden Multicore-System und verweisen abschließend auf ein aktuelles Forschungsprojekt in dem Gebiet.

4.1 Rekonfigurierbare Hardware

Die Idee, Hardware flexibel zu halten, geht auf G. Estrin zurück, der 1960 an der UCLA den Vorschlag machte, Computersysteme mit festen und variablen Anteilen zu gestalten.[7] Der variable Anteil kann der jeweiligen Anwendung angepasst werden. Bei Estrin waren die variablen Anteile Module für arithmetische Operationen, das Anpassen erfolgte manuell durch Einstecken und Entfernen von Modulen in ein sogenanntes Motherboard.



4 – Aufbau eines Field-Programmable Gate Array (FPGA)

Eine Hardwaretechnologie, die Estrins Ansatz automatisierbar machte, wurde erst Mitte der 80er Jahre von der Firma Xilinx mit dem Field Programmable Gate Array (FPGA) eingeführt. Ein FPGA ist ein Hardwarebaustein, der, wie in Abbildung 4 skizziert wird, aus drei Teilen besteht: aus einer Menge von

[7] Vgl. Gerald Estrin, „Reconfigurable Computer Origins: The UCLA Fixed-Plus-Variable (F+V) Structure Computer“, in: *IEEE Annals of the History of Computing* 24, 4 (2002), S. 3-9.

Logikelementen in einer matrixförmigen Anordnung, einem Verbindungsnetzwerk, in das die Logikelemente eingebettet sind, und externen Anschlüssen. Bei einem FPGA sind die Funktionen der Logikelemente und die konkreten Verbindungen zwischen den Logikelementen konfigurierbar. Die Funktion eines Logikelementes kann zum Beispiel eine Addition oder eine logische Verknüpfung der Eingänge sein. Die Konfiguration eines FPGA wird in SRAM-Speicherzellen abgelegt. Durch Schreiben dieser Speicherzellen, was je nach Größe des FPGAs von einigen Millisekunden bis zu wenigen hundert Millisekunden dauern kann, wird dem FPGA erst eine bestimmte Funktion eingeprägt.

FPGAs ermöglichen es, die Hardwarefunktion durch Schreiben von Speicherzellen zu verändern. Dies eröffnet die Möglichkeit, durch einen Softwareprozess die Logikebene zu verändern. Im Schichtenmodell eines Computersystems (siehe Abbildung 2) entspricht das einer Verschiebung der Grenze zwischen Hardware und Software von der Ebene des Instruktionssatzes nach unten in die Logikebene. Die tieferen Ebenen bleiben aber unangetastet. Das bedeutet insbesondere, dass auch bei FPGAs die mikroelektronische Schaltung mit ihren Transistoren bei der Fertigung endgültig festgelegt wird.

Zu Beginn der 90er Jahre wurde eine Reihe von Forschungsprojekten gestartet, um das Potenzial von FPGAs für den Aufbau von Computersystemen zu untersuchen. Ein prominentes Beispiel ist das DECPerLe-1 System[8], das am DEC Paris Research Lab entwickelt wurde. DECPerLe-1 konnte ausgewählte Anwendungen aus der Kryptografie rund drei Größenordnungen mal schneller ausführen als der schnellste Supercomputer zu der Zeit – und das bei einem Bruchteil der Kosten. Allerdings war die Programmierung der DECPerLe-1 sehr hardwarenahe und man benötigte dafür elektrotechnische Fachkenntnisse von Hardwareingenieuren.

Ab circa Mitte der 90er Jahre wurde die dynamische Rekonfiguration von FPGAs zum Forschungsthema. Bei der dynamischen Rekonfiguration wird ein FPGA zur Laufzeit neu rekonfiguriert, entweder der komplette Baustein oder auch nur ein Teil davon. Dynamische Rekonfiguration erlaubt es, die Hardwarefunktionen sehr rasch, d. h. im Bereich von Millisekunden, an neue Erfordernisse anzupassen. Ein 1997 im *Scientific American* erschienener Artikel stellte FPGAs und dynamische Rekonfiguration einer breiteren Leserschaft vor und präsentierte die Erkennung von Objekten in Bildern als Anwendungsbeispiel.[9] Durch Anpassung bzw. Spezialisierung der Hardwarefunktionen an die einzelnen zu suchenden Objekte konnte das System seine Verarbeitungsgeschwindigkeit deutlich steigern. Die spezialisierten Hardwarefunktionen

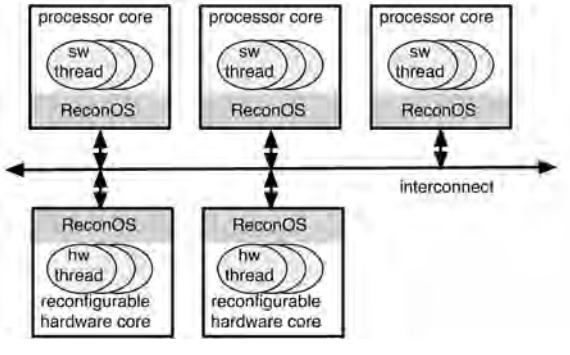
[8] Vgl. Patrice Bertin/Didier Roncin/Jean Vuillemin, „Programmable Active Memories: A Performance Assessment“, in: *Proceedings of the Conference on Field-Programmable Gate Arrays*, ACM, 1992.

[9] Vgl. John Villasenor/William H. Mangione-Smith, „Configurable Computing“, in: *Scientific American* (Juni 1997), S. 66-71.

wurden durch dynamische Rekonfiguration während der Laufzeit in das System eingebracht.

Anfang der 90er Jahre formierte sich das Forschungsgebiet Reconfigurable Computing, das Wissenschaftler aus der Informatik, der Elektrotechnik und vielen Anwendungsgebieten vereint. Heute ist Reconfigurable Computing ein etabliertes Gebiet mit einer Vielzahl von Konferenzreihen und zwei Fachzeitschriften. In den letzten 15 Jahren wurde eine Reihe von größeren Forschungsverbundprojekten durchgeführt, um die Nutzbarkeit von rekonfigurierbarer Hardware und speziell der dynamischen Rekonfiguration für Computing zu untersuchen. Beispiele dafür sind das DARPA-Programm „Adaptive Computing Systems“ (1997-2003) in den USA oder das Schwerpunktprogramm „Rekonfigurierbare Rechensysteme“ (2003-2009)[10] der DFG in Deutschland.

FPGAs sind seit ihrer Einführung ein großer wirtschaftlicher Erfolg und seit Jahren einer der am schnellsten wachsenden Sektoren der mikroelektronischen Industrie. Wegen ihres regelmäßigen Aufbaus eignen sich FPGAs sehr gut für mikroelektronische Fertigungsprozesse und durch die große Nachfrage können die Hersteller immer die aktuellsten und damit schnellsten Technologien für FPGAs nutzen. Heutige FPGAs haben riesige Logikkapazitäten und erlauben es, komplette Systeme mit mehreren Prozessoren, dedizierten Hardwarefunktionen, Speicher und Peripheriekomponenten auf einem Baustein in Form eines sogenannten Reconfigurable System-on-Chip unterzubringen.



5 – Hybrides Multicore-System

Eine große Herausforderung, an der wir seit einigen Jahren arbeiten, ist die Programmierung solcher Systeme. Wir entwickeln mit ReconOS[11] ein neuartiges Betriebssystem, das es erlaubt, sowohl die Softwarefunktionen als auch

[10] Vgl. Marco Platzner/Jürgen Teich/Norbert Wehn (Hg.), *Dynamically Reconfigurable Systems: Architectures, Design Methods and Applications*, Berlin, Heidelberg, 2010.

[11] Enno Lübbers/Marco Platzner, „ReconOS: Multithreaded Programming for Reconfigurable Computers“, in: *ACM Transactions on Embedded Computing Systems* 9, 1 (2009), S. 1-33.

die rekonfigurierbaren Hardwarefunktionen mit einem gemeinsamen Programmiermodell zu beschreiben. Dabei greifen wir auf Multi-Threading zurück, ein Programmiermodell, das durch den aktuellen Trend zu Multicore-Prozessoren sehr weit verbreitet und populär ist. In diesem Modell wird eine Anwendung durch eine Gruppe von unabhängigen Programmen, sogenannten Threads, beschrieben, welche grundsätzlich voneinander unabhängig sind und daher gleichzeitig auf verschiedenen Kernen eines Multi-Core-Prozessors ausgeführt werden können. Threads können über einen gemeinsamen Arbeitsspeicherbereich Daten austauschen und ihre Ausführung bei Bedarf mittels Betriebssystemfunktionen synchronisieren. Abbildung 5 zeigt ein Beispiel eines hybriden Multicore-Systems, das verschiedene Prozessor-Cores und rekonfigurierbare Hardware-Cores kombiniert. Hardwarefunktionen werden als Hardware-Threads auf den rekonfigurierbaren Hardware-Cores ausgeführt und kommunizieren und synchronisieren sich mit den Software-Threads mittels des Betriebssystems ReconOS.

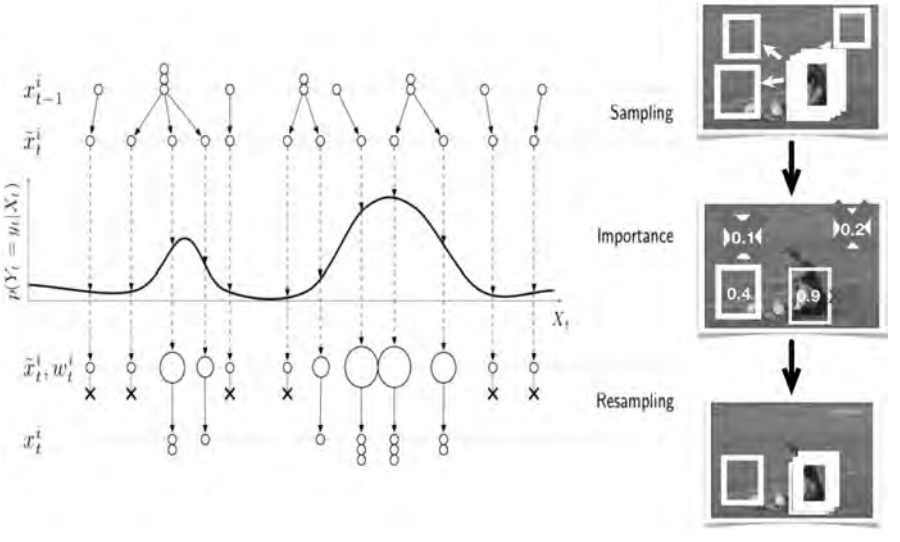
4.2 Selbst-adaptive Hybride Multi-Cores

6 – Beispiel zur Objektverfolgung in Videoströmen

Als Anwendungsbeispiel diskutieren wir einen selbst-adaptiven hybriden Multicore zur Objektverfolgung in Videoströmen.[12] Der Fußballspieler in Abbildung 6 wird markiert und soll im Laufe einer Videosequenz verfolgt werden. Das hier verwendete Verfahren zur Objektverfolgung ist ein sogenannter Partikelfilter, der Bild für Bild eine Anzahl von Schätzungen (Partikel) des Systemzustands (Ort, wo sich der Fußballspieler im Bild befindet) erzeugt. Der Systemzustand ist beschrieben durch die Koordinaten und Größe des Recht-

[12] Vgl. Markus Happe/Enno Lübbers/Marco Platzner, „A Self-Adaptive Heterogeneous Multi-Core Architecture for Embedded Real-Time Video Object Tracking“, in: *International Journal of Real-Time Image Processing. Special Issue*, Berlin, Heidelberg, 2011, S. 1-16.

ecks, das die Position des Fußballspielers umschreibt. Partikelfilter sind eine häufig angewendete Methode zur Online-Schätzung des Systemzustands eines nicht-linearen dynamischen Systems.

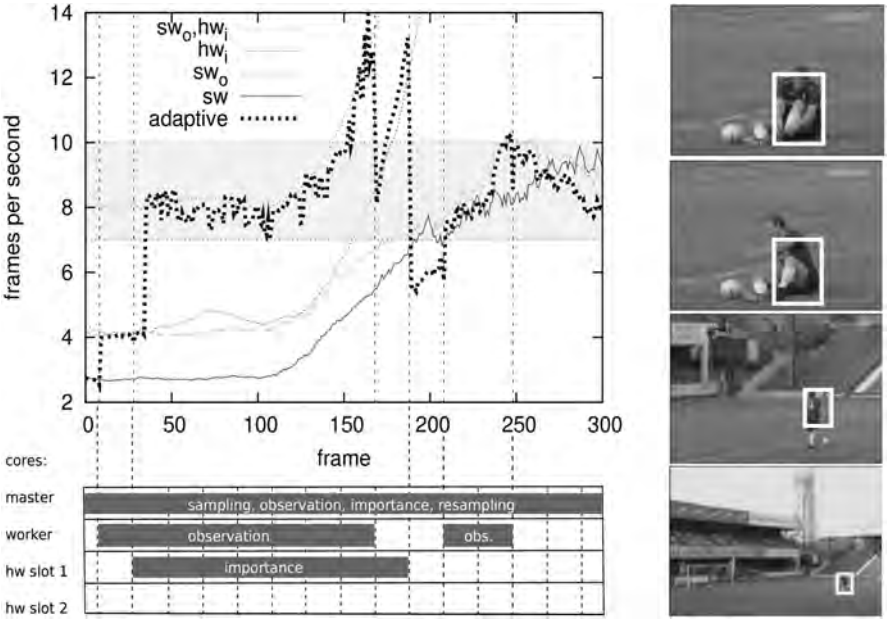


7 – Grafische Veranschaulichung der Schritte Sampling, Importance und Resampling eines Partikelfilters (links) und Beispiel-Bildsequenz (rechts)

Wie in Abbildung 7 gezeigt wird, läuft ein Partikelfilter als Iteration von drei Schritten ab. Im ersten Schritt (Sampling) werden ausgehend von der angenommenen Position des Spielers neue Schätzungen erzeugt. Dieser Schritt basiert auf einem mathematischen Zustandsmodell, das die aktuelle Bewegung des Spielers sowie ein gewisses Maß an Rauschen (Zufall) berücksichtigt. Im zweiten Schritt (Importance) werden die Schätzungen anhand der realen, gemessenen Daten aus dem nächsten Bild des Videos bewertet. Im vorliegenden Fall geschieht dies durch den Vergleich der Farbhistogramme von Schätzung und Messung. Verkürzt formuliert kann man sagen, dass bewertet wird, wie häufig die Farbe des T-Shirts des Spielers in den geschätzten Rechtecken vorhanden ist. Partikeln, in denen diese Farbe einen großen Anteil hat, wird eine hohe Importance zugewiesen, Partikeln mit niedrigen Farbanteilen eine niedrige. Die Importance kann als Wahrscheinlichkeit interpretiert werden, dass die Schätzung tatsächlich das gesuchte Objekt abbildet. Der Partikel mit der höchsten Importance ist die aktuelle Ausgabe des Verfahrens. Im letzten Schritt (Resampling) werden gute Schätzungen vervielfacht und schlechte Schätzungen gelöscht.

Der Partikelfilter wurde auf einem hybriden Multicore in einem FPGA implementiert. Um eine parallele Verarbeitung auf mehreren Kernen zu ermöglichen, wurde die Anwendung in drei Threads aufgespalten, welche die

Phasen Sampling, Importance und Resampling realisieren. Wie in Abbildung 8 unten gezeigt wird, besteht das System aus zwei Prozessoren (Master, Worker) und zwei Hardware Cores (HW Slot 1, HW Slot 2). Das Ziel ist, die Performance, gemessen in der Anzahl der bearbeiteten Bilder pro Sekunde (frames per second), innerhalb eines vorgegebenen Bereichs von 7 bis 10 Bildern pro Sekunde zu halten. Zu Beginn verwendet das System nur einen Prozessor und erzielt damit eine sehr niedrige Performance von unter 3 Bildern pro Sekunde. Daraufhin schaltet das System autonom zuerst einen zweiten Prozessor und dann einen Hardware-Core dazu. Als Resultat steigt die Performance auf ca. 8 Bilder pro Sekunde. Da der Fußballspieler im Laufe der Videosequenz nach hinten läuft (siehe Abbildung 8 rechts) und durch die kleineren Rechtecke der Rechenaufwand für den Partikelfilter sinkt, übersteigt die Performance bald den Zielbereich. Als Reaktion darauf schaltet das System zuerst den zweiten Prozessor und dann den Hardware-Core ab. Da dadurch die Performance aber zu stark sinkt, wird der zweite Prozessor für eine kurze Zeit wieder dazu genommen.



8 – Aktivitäten der einzelnen Prozessor- und Hardware-Cores (unten), Bildsequenz (rechts) und Performance (oben) des selbst-adaptiven Multicore-Systems

Dieses Beispiel demonstriert zwei wesentliche Punkte: Erstens zeigt es, dass mit moderner FPGA-Technologie Funktionen zur Laufzeit zwischen Hardware und Software migriert werden können und somit eine dynamische Anpassung der Hardware/Software-Grenze ermöglicht wird. Zweitens kann die-

ses System als selbst-adaptiv bezeichnet werden, da die Adaption, d. h. das Aktivieren und Deaktivieren von Rechenressourcen, vom System selbst ohne Eingriff von außen gesteuert wird. Der Algorithmus jedoch, nach dem das System diese Entscheidungen trifft, wurde bei der Entwicklung festgelegt.

4.3 Von Selbst-Adaption zu Self-Awareness

Eine Weiterentwicklung von selbst-adaptiven Systemen führen wir im Rahmen des EU-Projekts EPiCS[13] durch. In EPiCS studieren wir sogenannte propriozeptive Computersysteme. Darunter verstehen wir Systeme, die durch geeignete Sensoren nicht nur ihre Umgebung, sondern auch ihren inneren Zustand erfassen können. Für ein hybrides Multicore-System zur Objektverfolgung in Videos wäre die Umgebung beispielsweise ankommende Videodaten oder die Benutzerinteraktion zum initialen Auswählen des Objekts. Der innere Zustand könnte beispielsweise die aktuelle Ressourcenauslastung und Temperaturverteilung oder entdeckte Fehlfunktionen umfassen.

In Anlehnung an die Psychologie bezeichnen wir im Kontext des EPiCS-Projekts das Wissen und die Modelle über externe und interne Ereignisse und Zustände als Self-Awareness. Basierend auf diesem Wissen entscheidet das System autonom über mögliche Reaktionen. In unserem Partikelfilterbeispiel wären mögliche Reaktionen das Aktivieren und Deaktivieren von Cores oder eine Reduktion der Anzahl der Partikel, was die Rechenlast für den Preis einer etwas schlechteren Schätzung verringert. Die Auswahl der entsprechenden Reaktion bezeichnen wir als Self-Expression.

Unser langfristiges Ziel ist es, Computersysteme zu entwerfen, die Parameter wie Performance, Ressourcennutzung und Energieeffizienz, aber auch Eigenschaften wie Zuverlässigkeit und Sicherheit autonom zur Laufzeit optimieren können. Ein wesentlicher Schritt dabei ist das Erzeugen von Modellen der Self-Awareness und Self-Expression. Diese Modelle dienen zum einen zur Beschreibung und Diskussion der Begrifflichkeiten und zum anderen als Schritt im systematischen Entwurf von Computersystemen. Tabelle 2 zeigt zum Beispiel eine aus der Psychologie abgeleitete Klassifikation[14], die fünf unterschiedliche Ebenen der Self-Awareness in Computersystemen beschreibt. Die niedrigste Form von Self-Awareness ist demnach Stimulus-Awareness, wie sie ein einfaches passives Stimulus-Response-System aufweist. Das System besitzt kein Wissen über die Ursachen der Stimuli und keine Erinnerung an vergangene Ereignisse und kann damit auch keine Vorhersagen über zukünftige Ereignisse machen. Die nächste Ebene ist Interaction-Awareness, die aktive Kommunikation in einem vernetzten System voraussetzt. Darüber be-

---

[13] *EPiCS: ENGINEERING PROPRIOCEPTION IN COMPUTING SYSTEMS*, online unter: http://www.epics-project.eu, zuletzt aufgerufen am 24.04.2014.

[14] Vgl. Ulric Neisser, „The Roots of Self-Knowledge: Perceiving Self, It, and Thou“, in: *Annals of the New York Academy of Sciences* 818 (1997), S. 19-33.

findet sich Time-Awareness, die ein Langzeitgedächtnis voraussetzt und damit Lernen und Voraussagen ermöglicht. Die Ebene Goal-Awareness zeichnet sich dadurch aus, dass das System Wissen über seine Ziele und eventuelle Randbedingungen besitzt und den Stand der Erreichung feststellen kann. Die höchste Ebene ist schließlich die der Meta-Self-Awareness, bei der ein System Wissen über seine eigene Self-Awareness erlangt und Nutzen/Kosten seiner Self-Awareness abschätzt.

Tabelle 2: Ebenen der Self-Awareness nach Becker et al.[15]

Ebene	**Charakteristik**
stimulus-aware	Stimulus-Response System, passiv
interaction-aware	vernetztes System, aktiv
time-aware	Langzeitgedächtnis, Lernen, Vorhersage
goal-aware	Ziel bekannt, Auswahl aus mehreren Strategien
meta-self-aware	„Philosophie-Agent“

Konkrete Anwendungen von Computersystemen verwenden meistens mehrere Ebenen der Self-Awareness. Das im vorigen Abschnitt präsentierte hybride Multicore-System ist definitiv *time-aware*, da das Verfahren des Partikelfilters ein Systemmodell verwendet, um Vorhersagen zu machen. Nachdem die vorgestellte Implementierung auch ihr Performanceziel explizit kennt und laufend Maßnahmen zur Erreichung des Ziels durchführt, kann man das System auch als *goal-aware* klassifizieren. Computersysteme auf der Ebene der Meta-Self-Awareness werden im EPiCS-Projekt derzeit nicht untersucht. Man könnte solche Systeme am ehesten als „Philosophie-Agenten“ beschreiben, die ihre Awareness reflektieren.

[15] Tobias Becker/Andreas Agne/Peter R. Lewis et al., „Engineering Proprioception in Computing Systems“, in: *Proceedings of the Conference on Embedded and Ubiquitous Computing (EUC)*, IEEE, 2012.

5. Conclusion

Die Grenze zwischen Hardware und Software ist durch neue Hardwaretechnologien im Laufe der Zeit durchlässig geworden. Rekonfigurierbare Hardware erlaubt heute die Konstruktion von selbst-adaptiven Systemen, die sich laufend veränderten Anforderungen anpassen können. Besonders bei komplexen Systemen, die in vielfältiger Weise mit anderen Computersystemen und Benutzern interagieren, lassen sich viele Entscheidungen nicht mehr zur Entwurfszeit vorwegnehmen und damit fest in das System programmieren. Hier ist die Selbst-Adaption ein vielversprechender Ansatz. Die Auflösung von starren Grenzen zwischen Hardware und Software wirft aber auch eine Reihe von neuen Fragestellungen auf. Aus technischer Sicht sind das neben der Programmierbarkeit solcher Systeme vor allem die Modellierung und die Validierung. Bei der Modellierung geht es um die klare konzeptionelle Beschreibung der Komponenten von selbst-adaptiven Systemen und deren Funktionalität. Heute entstammen die meisten Beispiele selbst-adaptiver Hardware/Software-Systeme einem Ad-hoc-Entwurf. Erst eine umfassende, aber nachvollziehbare Modellierung von Selbst-Adaption ermöglicht einen systematischen Entwurfsprozess und den Vergleich mit klassischen Systemen. Ein Teil der Modellierung ist auch die Bereitstellung von geeigneten algorithmischen Methoden für die Entscheidungsfindung und das Lernen. Bei der Validierung von Hardware oder Software möchte man bestimmte Systemeigenschaften zur Entwurfszeit nachweisen. Wie kann man aber zum Beispiel die Sicherheit eines Systems nachweisen, das sich selbständig verändern kann? Neben dem technisch-wissenschaftlichen Bereich hat die sich auflösende Grenze zwischen Hardware und Software auch Auswirkungen auf die Ausbildung. Die Beschäftigung mit selbst-adaptiven Computersystemen wird vermehrt Experten des Computer-Engineering erfordern, die in Informatik und Elektrotechnik gleichermaßen geschult sind.

Literatur

Becker, Tobias/Agne, Andreas/Lewis, Peter R. et al., „Engineering Proprioception in Computing Systems“, in: *Proceedings of the Conference on Embedded and Ubiquitous Computing (EUC)*, IEEE, 2012.

Bertin, Patrice/Roncin, Didier/Vuillemin, Jean, „Programmable Active memories: A Performance Assessment“, in: *Proceedings of the Conference on Field-programmable Gate Arrays*, ACM, 1992.

Ceruzzi, Paul E., *A History of Modern Computing*, Cambridge, MA, 2003.

Christensen, Clayton M., *The Innovator's Dilemma*, Boston, MA, 1997.

EPiCS: ENGINEERING PROPRIOCEPTION IN COMPUTING SYSTEMS, online unter: http://www.epics-project.eu, zuletzt aufgerufen am 24.04.2014.

Estrin, Gerald, „Reconfigurable Computer Origins: The UCLA Fixed-Plus-Variable (F+V) Structure Computer", in: *IEEE Annals of the History of Computing* 24, 4 (2002), S. 3-9.

Happe, Markus/Lübbers, Enno/Platzner, Marco, „A Self-Adaptive Heterogeneous Multi-Core Architecture for Embedded Real-Time Video Object Tracking", in: *International Journal of Real-Time Image Processing. Special Issue*, Berlin, Heidelberg, 2011, S. 1-16.

Hennessy, John/Jouppi, Norman/Przybylski, Steven et al. „MIPS: A Microprocessor Architecture", in: *Proceedings of the 15th Workshop on Microprogramming (MICRO)*, Piscataway, NJ, 1982, S. 17-22.

Lübbers, Enno/Platzner, Marco, „ReconOS: Multithreaded Programming for Reconfigurable Computers", in: *ACM Transactions on Embedded Computing Systems* 9, 1 (2009), S. 1-33.

Neisser, Ulric, „The Roots of Self-Knowledge: Perceiving Self, It, and Thou", in: *Annals of the New York Academy of Sciences* 818 (1997), S. 19-33.

On-The-Fly Computing, online unter: http://sfb901.uni-paderborn.de, zuletzt aufgerufen am 24.04.2014.

Patterson, David A., „Reduced Instruction Set Computers", in: *Communications of the ACM* 28, 1 (1985), S. 8-21.

Platzner, Marco/Teich, Jürgen/When, Norbert (Hg.), *Dynamically Reconfigurable Systems: Architectures, Design Methods and Applications*, Berlin, Heidelberg, 2010.

Teich, Jürgen/Haubelt, Christian, *Digitale Hardware/Software-Systeme. Synthese und Optimierung*, 2. Aufl., Berlin, Heidelberg, New York, NY, 2007.

Villasenor, John/Mangione-Smith, William H., „Configurable Computing", in: *Scientific American* (Juni 1997), S. 66-71.