Hardware oder Software?

Eine Entscheidung beim Entwerfen von Embedded Systems

Prof. Dr. Wolfgang Matthes
Peukinger Weg 34
59423 Unna
http://www.realcomputerarchitecture.com

> Zum Philosophieren sind die zwei ersten Erfordernisse diese: erstlich, daß man den Mut habe, keine Frage auf dem Herzen zu behalten, und zweitens, daß man alles das, *was sich von selbst versteht*, sich zum deutlichen Bewußtsein bringe, um es als Problem aufzufassen.
>
> *Arthur Schopenhauer*

Der Ingenieur steht gelegentlich vor diese Frage, wenn er beginnt, eine Entwicklungsaufgabe zu lösen. Im strengen Sinne gibt es die Alternative aber gar nicht, da jede Software eine Hardware-Grundlage braucht, um laufen zu können. Im praktischen Sinne handelt es sich darum, zu entscheiden, ob man seine Aufgabe auf Grundlage einer fertigen Hardware-Plattform (also "nur" durch Software (Programmierung)) lösen oder ob man eigene Hardware entwickeln soll – von einfachen Anpassungs- und Zusatzschaltungen bis hin zum vollständigen System "aus einem Guß".

Professionelles Entwickeln muß Ergebnisse hervorbringen, die im Wettbewerb bestehen können. Dies sollte genügen, um auf die Gefahren eines schematischen, letzten Endes gedankenlosen Herangehens ("ist doch selbstverständlich", "haben wir immer schon so gemacht") hinzuweisen. Betrachten wir zwei Szenarien:

- Sie schnitzen eine Hardware aus dem Vollen, müssen den Entwurf ein paarmal überarbeiten, weil Schaltkreise, obwohl mit riesigem Werbegetöse angekündigt, nach Lieferung der ersten Muster sang- und klanglos gestrichen wurden, fallen zweimal durch den EMV-Test und ernähren zeitweise eine Anwaltskanzlei, um Angriffe von Inhabern fremder Patentrechte abzuweisen. Ihr Konkurrent nimmt einen Industrie-PC, schreibt ein C++-Programm, das unter der aktuellen Windows-Version läuft, ist ein halbes Jahr eher am Markt, und muß sich weder um die EMV-Verordnung, noch um das Patentrecht, noch um die Ersatzteilversorgung kümmern.
- Sie kaufen die neueste PC-Plattform mit dem gigantischsten Betriebssystem und entwickeln in der neumodischsten Sprache. Es will und will nur nicht recht laufen. Zudem stoßen Sie auf ein paar "undocumented features" im Systemkern, die sich in Ihrer Anwendung aber als handfeste Bugs bemerkbar machen. Ihr Konkurrent nimmt einen Wald- und Wiesen-Mikrocontroller, schaltet ihn mit ein paar Speicher-ICs und einem "selbstgeschmiedeten" FPGA zusammen – und hat eine Lösung, die 1/10 des Stromes verbraucht, wenigstens die 10-fache MTBF hat, keine mechanisch bewegten Teile enthält und auch in einem Billiggehäuse die EMV-Anforderungen erfüllt. Dabei hängt er nicht von fremden, unbeeinflußbaren proprietären Standards ab, kann jeden Funktionsfehler selbst beseitigen und kann guten Gewissens zusagen, daß die Ersatzteilversorgung über mehrere Jahre hinweg gesichert sein wird.

Offensichtlich ”kommt es darauf an” - auf das Marktsegment, auf die Preisklasse, auf die jeweils besonderen Umstände und Anforderungen. Da Entscheidungen der hier in Rede stehenden Art schnell und treffsicher gefällt werden müssen, ist es von Vorteil, hierfür objektive Orientierungskriterien zu haben.

Zum Stand der Technik

- An die Funktionseigenschaften einer Neuentwicklung werden hohe Anforderungen gestellt. (Damit, daß die jeweilige Grundfunktion “gerade so” erfüllt wird, dürfte sich wohl kein Kunde mehr zufriedengeben.)
- Das Programmier-Paradigma hat sich weitgehend durchgesetzt. Somit wird praktisch jedem Lösungsansatz für auch nur halbwegs komplexe funktionelle Anforderungen irgendeine Form der Programmierbarkeit zugrunde gelegt. (Kaum noch jemand baut hochkomplizierte Spezialschaltungen, die nicht irgendwie programmierbar bzw. durch Programmierung steuerbar sind.)

Es geht also in der Praxis nicht um den extremen Gegensatz: fertige Systemumgebung oder Einzweck-Spezialschaltung, sondern um kostenoptimierte Verbundlösungen.

Das Entscheidungsproblem (Hardware oder Software) stellt sich in der Praxis dann, wenn:

- extreme Realzeitanforderungen zu erfüllen sind,
- ein Funktionsumfang verwirklicht werden soll, der im jeweiligen Kostenrahmen bisher nicht implementiert werden konnte,
- Entwicklungszeiten verringert werden sollen (Time to Market).

Weshalb ersetzen wir Hardware durch Software?

- Der herkömmliche Trend seit den 60er Jahren.
- Rückgriff auf ein höheres Niveau der Zuhandenheit (“weniger selbst tun”).
- Zuhandenheit hochleistungsfähiger Technologien (Taktfrequenzen).
- Verkürzung der Entwicklungszeit (“es ist nur zu programmieren” = Time to Market).
- Verringerung der Hardwarekosten (durch Einsatz von Hardware aus der Massenfertigung).
- Beherrschung von Kompliziertheit (durch Zerlegen in handliche Programmstücke).
- Ersparen der Hardware-Entwicklung (statt dessen Rückgriff auf zuhandene funktionstüchtige Plattformen).
- Kurze Änderungszyklen.
- Flexible und komfortable Entwicklungswerkzeuge.
- Funktionelle Flexibilität (es läßt sich “alles” programmieren – und wir *dürfen* programmieren).
- Software kann nicht ausfallen (verschleißen)[1].

Weshalb ersetzen wir Software durch Hardware?

- Ein (wieder mal) neuerer Trend – angeregt durch Verfügbarkeit hochintegrierter programmierbarer Schaltkreise (FPGAs, CPLDs).

1: Gleichwohl kann Software Fehler enthalten (und auch veralten – wenn es weit und breit keine Hardware-Plattform mehr gibt, auf der sie laufen könnte).

- Mehr Leistung.
- Kontrolle über den gesamten Lebenszyklus (nicht mehr auf Zulieferungen, fremde Standards und fremde Schutzrechte angewiesen sein).
- Alle Funktionsfehler sind unsere Fehler. Das mag gelegentlich das Selbstbewußtsein beeinträchtigen, hat aber für sich, daß wir sie auch beseitigen *dürfen*.
- Kostenoptimierung = Verbilligung der Hardware-Plattform, kein Software-Overhead (besserer Wirkungsgrad).
- Lösung der Aufgabe mit weniger Silizium, Strom, EMI (langsamst-mögliche Taktierung).
- Besseres Realzeitverhalten.
- Kürzere Latenzzeiten.
- Echter Parallelismus (der dem Problem überhaupt inhärente Parallelismus kann voll ausgenutzt werden, Schaltungsstruktur $\triangleq$ Problemstruktur $\triangleq$ Datenfluß, höchster Wirkungsgrad).
- Unmittelbare (evidente) Vergegenständlichung der Informationswandlungen (wir *müssen nicht* programmieren). Programme sind gelegentlich unangemessen und unübersichtlich ("semantische Lücke" zwischen Problembeschreibung und Programmablauf, zuviel Overhead).

Analyse der Entwurfsaufgabe

Es gilt, sich über zwei Problempunkte Klarheit zu verschaffen:

- Über die Funktionalität: was ist eigentlich zu tun? (die auszuführenden Informationswandlungen).
- Über das Realzeitraster: in welcher Zeit sind die Informationswandlungen auszuführen?

Wir müssen die Anforderungen und Lösungsmöglichkeiten wenigstens in grober Näherung sozusagen vorsortieren und bewerten (Rückgriff auf Erfahrungen, Analogieschlüsse (zu bereits bekannten und bewährten Lösungen), Probeentwürfe und -programme, Simulation).

Das Realzeitraster

Unter diesem Begriff wollen wir alle Anforderungen an das zeitliche Verhalten zusammenfassen. Solche Anforderungen betreffen Gesamt-Ausführungszeiten, Latenzzeiten (von der auslösenden Anforderung bis zum Beginn des Reagierens), Takt-Zykluszeiten usw. Wir wollen zunächst annehmen, daß es eine "kritische" Zeitvorgabe t gibt (Tabelle 1). Die Aufgabe der Systementwicklung besteht dann darin, zu gewährleisten, daß die jeweils geforderten Informationswandlungen in der vorgegebenen Zeit t mit kostenoptimalen Mitteln erledigt werden (Kosten über alles = über den gesamten Lebenszyklus = Total Cost of Ownership). Je geringer die Zeitvorgabe t und je größer die Kompliziertheit bzw. Komplexität, um so größer der Aufwand.

Hinweis: Komplexität und Kompliziertheit sind keine Wechselworte!

Komplexität beschreibt Ressourcen-Anforderungen des Algorithmus in Abhängigkeit von der Problemgröße. Beschreibung hat die Form O (n) (Order of... = Größenordnung von...). *Komplex* sind mehr als linear mit der Problemgröße wachsende Anforderungen (z. B. an Verarbeitungsleistung oder Speicherplatz).

Kompliziertheit = Spitzfindigkeit, Verwickeltheit, Vielfalt der Funktionen. *Kompliziert* = verwickelt (sophisticated). Läßt sich näherungsweise durch Umfang der Problembeschreibung (= funktionelle Spezifikation) kennzeichnen.

Zeitvorgabe (t)	Beispiele	technische Mittel
10 ns	Speicheransteuerung, Videodarstellung, Befehlsausführung in Prozessoren	Hardware auf Gatter-Ebene (auch: Transistor-Ebene)
100 ns	Gerätesteuerung	Hardware auf Gatter-Ebene, State Machines, Mikroprogrammierung
1 µs	Gerätesteuerung	Hardware auf Gatter-Ebene, State Machines, Mikroprogrammierung, Maschinenprogrammierung (was sich mit 10...50 Maschinenbefehlen erledigen läßt)
1 ms	Gerätesteuerung, Regelungsaufgaben (Closed Loop)	Realzeit-Software (was sich mit wenigen tausend Maschinenbefehlen erledigen läßt)
10 ms	Gerätesteuerung, Regelungsaufgaben (Closed Loop), Bedienung/Anzeige	Realzeit-Software
100 ms	Regelungsaufgaben (Closed Loop), Bedienung/Anzeige, Prozeßsteuerung, Informationsbeschaffung (Retrieval)	Komplexe Realzeit-Software, Vernetzung
1 s	Prozeßsteuerung, Informationsbeschaffung (Retrieval)	Realzeit- bzw. beliebige Software (je nach Hardware-Plattform)
Kommt nicht drauf an	Fertigungssteuerung, allgemeine Verwaltung	Beliebige Software (auch Windows etc.), Vernetzung (incl. Internet)

Tabelle 1 Größenordnungen des Realzeitrasters.

Die einfachste Vor-Entscheidung:

- Wenn t groß: das kostengünstigste zuhandene Mittel aussuchen: reicht es aus?
- Wenn t extrem klein: ist die Aufgabe überhaupt realisierbar? Es kommen nur schaltungstechnische Lösungen in Frage; gelegentlich sind erfinderische Bemühungen notwendig.

Auf welcher Ebene der Zuhandenheit aufsetzen?
Die Antwort hängt von der Realzeitraster-Vorgabe und vom Stand der Technik ab:

- < 1 ns: Basis- (Halbleiter-) Technologie,
- wenige ns: Fertigungstechnologie (einschließlich zuhandener Halbleiter-Technologien),
- < 1 µs: Bauelemente-Technologie (zuhandene Schaltkreise),
- vom ms-Bereich an: zuhandene Funktionseinheiten...fertige Systemumgebungen.

Kostenoptimierung = nicht zuviel Overkill für die Problemlösung = gerade soviel, um Reserven für Änderungen zu haben (den gesamten Lebenszyklus bedenken!).

Faustregel:
Die Entscheidung – *Hardware oder Software* – ist beim Stand der Technik nur im Bereich des Realzeitrasters von 100 ns bis 100 ms von Bedeutung.

Implementierungseffizienz
Es liegt nahe, verschiedene Auslegungen der Hardware (sowohl selbst zu bauende als auch – zwecks Laufenlassen von Software – fertig zu beschaffende) daraufhin zu untersuchen, ob sie als Plattform zur Implementierung der jeweiligen Informationswandlungen (Algorithmen) vom Leistungsvermögen her ausreichend und von den Kosten her annehmbar sind. Wir wollen aber andersherum herangehen – nämlich den *Algorithmus* daraufhin untersuchen, ob er sich eher dazu eignet, in spezieller Hardware vergegenständlicht oder eher dazu, als Programm hingeschrieben zu werden.

Was ist eigentlich die ideale technische Vergegenständlichung eines Algorithmus? – Doch wohl eine Anordnung, die aus den Eingangsdaten (Argumenten) die Resultate unmittelbar, sozusagen auf einen Schlag bildet (d. h. durch unmittelbare Zuordnung und nicht durch zeitliche Folgen aufeinanderfolgender Verarbeitungschritte). Abb. 1 zeigt das geradezu triviale Blockschaltbild. Offensichtlich läßt sich so etwas nur dann bauen, wenn die Anzahl der Bits (bzw. Leitungen) nicht allzu hoch ist und wenn die Verknüpfungen nicht allzu kompliziert sind. Ebenso offensichtlich ist ein Ausweg: die abschnittsweise Verknüpfung (Abb. 2). Eine solche Anordnung arbeitet taktgesteuert, wobei in jedem Takt aus den jeweils ausgewählten Argument-Abschnitten die jeweiligen Ergebnis-Abschnitte gebildet werden. Die höchste überhaupt mögliche Verarbeitungsleistung einer solchen Anordnung wird dann erreicht, wenn in jedem Maschinentakt ein Abschnitt des Ergebnisses gebildet werden kann, dessen Bitanzahl der Verarbeitungsbreite der Anordnung entspricht bzw. wenn es ausreicht, zur Bildung des Ergebnisses auf jeden Argument-Abschnitt genau einmal zuzugreifen. Ob dies überhaupt gelingen kann, ist letzten Endes eine Eigenschaft des Algorithmus.

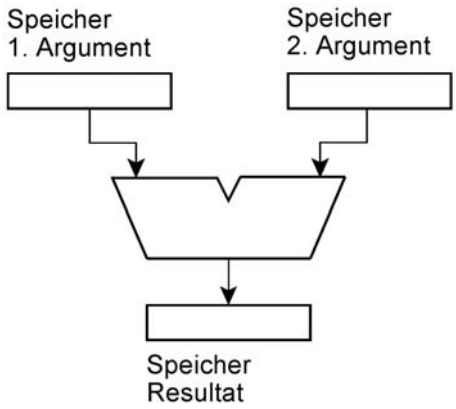

Abb. 1 Ausführung eines Algorithmus durch direkte Zuordnung.

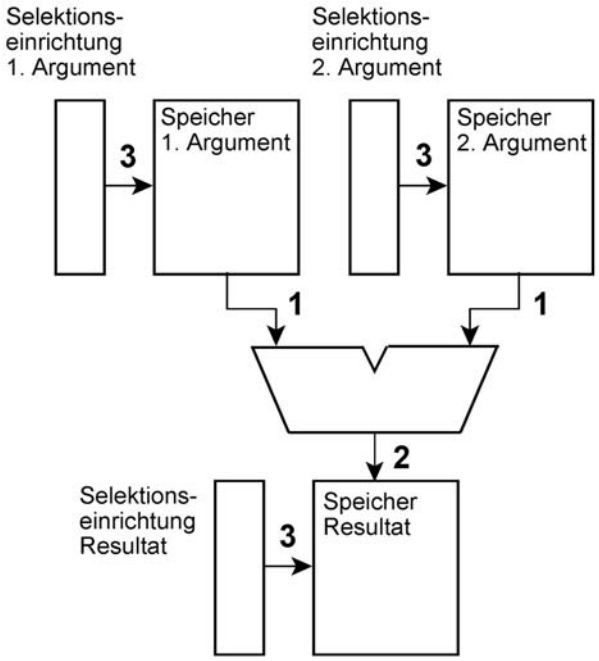

Abb. 2 Abschnittsweise Ausführung eines Algorithmus. 1 - Informationsleitungen für Argumentabschnitte; 2 - desgl. für Resultatabschnitt; 3 - Auswahlleitungen (Speicheradressen).

1. Beispiel (wo es klappt): die Bildung des Skalarprodukts zweier Vektoren.

2. Beispiel (wo es im allgemeinen Fall nicht klappt): Sortierabläufe. Hierbei ist es praktisch nicht zu vermeiden, auf die einzelnen Argument-Abschnitte mehrmals zuzugreifen.

Diese Eigenschaft des Algorithmus läßt sich durch eine Zahlenangabe ausdrücken, die wir als *Implementierungseffizienz* e_i bezeichnen wollen.

$$e_i = \frac{\sum_{i=1}^{n} CARDB(A_i) + \sum_{j=1}^{m} CARDB(R_j)}{z \cdot (ARG\_LINES + RES\_LINES)}$$

Zur Bedeutung der Symbole:

- CARDB (A_i), CARDB (R_j): Anzahl der Bits, mit denen die Argumente und Resultate repräsentiert (bzw. codiert) werden.
- ARG-LINES, RES-LINES: Anzahl der Signalleitungen, die zum Transportieren der Argument- und Resultatabschnitte zur Verfügung stehen (Stichwort: Verarbeitungsbreite),
- z: Anzahl der Maschinenzustände (Taktzyklen), die benötigt werden, um alle Resultate zu bilden ($z \geq 1$).

Ist eine technisch gegebene Leitungszahl größer als die Anzahl der Bits des betreffenden Abschnittes, ist die Abschnittslänge anstelle der Leitungszahl einzusetzen.

Die Implementierungseffizienz e_i ist somit eine dimensionslose Zahl im Intervall $0 < e_i \leq 1$. e_i ist gleich 1, wenn die beschriebene zweckmäßigste Implementierung tatsächlich möglich ist, d. h., wenn es gelingt, eine Anordnung zu bauen, die in jedem Taktzyklus entsprechend ihrer Verarbeitungsbreite zum gewünschten Endergebnis beiträgt. Dies kann nur dann erreicht werden, wenn für jeden Abschnitt des Resultats gilt, daß dieser durch Zuordnung (im Sinne von Abb. 1) aus den jeweils ausgewählten Abschnitten der Argumente gebildet werden kann, wobei in diese Zuordnung höchstens noch Zustands-Angaben einbezogen werden, die eindeutig aus den zuvor verarbeiteten Abschnitten bestimmt worden sind (Abb. 3).

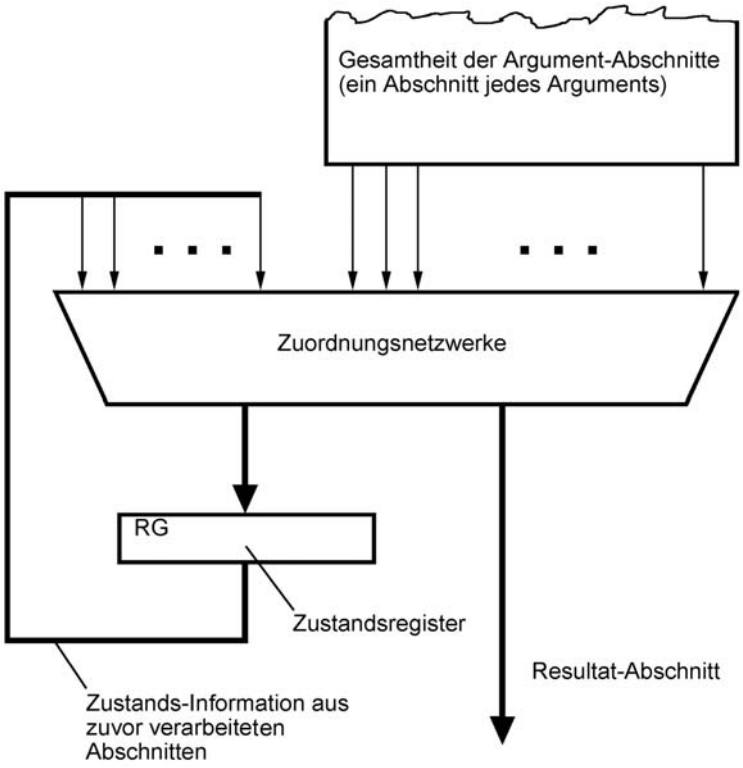

Abb. 3 Allgemeines Schema der Ausführung eines Algorithmus mit $e_i = 1$.

Es ist durchaus möglich, diesen Ansatz bis in die Feinheiten hinein zu formalisieren. Um zu Beginn der Bearbeitung eines Projekts Entscheidungen zu treffen, reicht aber typischerweise eine "heuristische" Anwendung aus. Hierbei ist zu untersuchen, ob die Problemstellung auf Algorithmen führt, die eine Implementierungseffizienz von 1 (oder nahezu 1) haben oder ob diese ohnehin deutlich geringer ausfällt.

Praxistip: Man versuche, für die leistungsbestimmenden Informationswandlungen Probeentwürfe im Sinne von Abb. 3 auszuführen. Ein "gröberes" Register-Transfer-Niveau reicht vollkommen aus (erforderliche Breite der Signalwege, Abschätzung der Aufwendungen für die kombinatorischen Zuordnungs-Netzwerke usw.). Es wird sich recht schnell zeigen, ob so etwas überhaupt durchführbar ist oder nicht (Aufwand zu hoch, Steuerung zu kompliziert, Ergebnisse lassen sich grundsätzlich nicht auf gewünschte Weise – d. h. mit nur einmaligem Zugriff auf jede Informationsstruktur – bilden).

Je näher e_i an Eins liegt, um so eher lohnt es sich, an eine Hardwarelösung zu denken (auch in dem Sinne, eine Prozessorstruktur durch Zusatzbeschaltung zu erweitern).

Bei an sich kleinem e_i ist es typischerweise besser, Hardware-Entwicklungen zu vermeiden. Statt dessen die jeweils leistungsfähigste Software-Plattform aussuchen.

Die Implementierungseffizienz ist < 1, wenn

- die abschnittsweise Zuordnung technisch nicht verwirklicht werden kann (Aufwand, Kompliziertheit), so daß Folgen mehrerer Taktzyklen (Maschinenzustände) nötig sind, um jeweils einen Resultatabschnitt zu bilden,
- Resultatbits von Argumentbits abhängen, die sich in verschiedenen Abschnitten befinden können, so daß Zugriffe auf mehrere Argumentabschnitte notwendig sind, um diese Resultatbits zu bestimmen,
- gewisse Argumentbelegungen zur Folge haben, daß Teile bereits gebildeter Resultatabschnitte geändert werden müssen (dies erfordert, erneut auf diese Abschnitte zuzugreifen) bzw. daß man das Ergebnis gar nicht in abschnittsweise liefern kann, sondern daß zunächst Zwischenergebnisse zu bilden sind.

Ein Sachverhalt nach Punkt 1 ist nicht immer ein unüberwindliches Hindernis: es ist letztlich eine Ermessensfrage, was man als “zu aufwendig” oder “zu kompliziert” ansieht. Hingegen bezeichnen die Punkte 2 und 3 objektive Grenzen. Solche Algorithmen können nie mit e_i=1 implementiert werden (auch dann nicht, wenn der Aufwand keine Rolle spielt). Ein plausibles Beispiel ist das Umordnen von Bits in einem größeren Bitfeld (z. B. von Pixeln in einem Bildspeicher) gemäß den Angaben einer Indexliste. Jedes Argumentbit kann hierbei grundsätzlich in jede Resultatposition transportiert werden, so daß es notwendig sein kann, bereits gebildete Resultatabschnitte nochmals aufzurufen, um neue Werte einzufügen.

Leistungsvergleich

Sind die beliebten MIPS- bzw. MFLOPS-Leistungsangaben schon beim Vergleichen von Prozessoren verschiedenartiger Architektur nicht immer aussagekräftig, so sind praktisch gar nicht brauchbar, wenn Hardware- und Softwarelösungen miteinander verglichen werden sollen. (Beispiel: eine Spezialhardware braucht, angeregt durch wenige “Befehle” (zur Parameterübergabe und Ablauf-Auslösung), für eine bestimmte Aufgabe 1 µs. Ein universeller Prozessor müßte, um dasselbe zu leisten, hierzu 100 Befehle ausführen. Demzufolge müßte ein Prozessor mit 100 MIPS eingesetzt werden, um die Geschwindigkeit der Sonderhardware zu erreichen.)

Brauchbare Leistungskennwerte (die sich oftmals auch mit hinreichender Genauigkeit abschätzen lassen) sind:

- Die Latenz- und Ausführungszeiten.
- Die Verarbeitungsleistung in Nutz-Bits/s. Wir betrachten hierzu lediglich die Informationstransporte, die tatsächlich zur Problemlösung beitragen (also Argument- oder Resultatbits betreffen). Der Grundgedanke: eine jeweils ideale Maschine würde nur Argumente holen und Resultate abliefern. Wird hingegen die verfügbare “Bandbreite” der Maschine (im Sinne der Datenraten bei Speicher- und Buszugriffen) überwiegend dafür genutzt, Befehle zu holen, Cache-Einträge umzuschaufeln usw., so mögen die vielen MBits/s zwar im Prospekt etwas hermachen, sie tragen aber zunächst nichts zur Problemlösung bei).

Praxistips:

- Software: die leistungsbestimmenden innersten Schleifen probeweise ausprogrammieren,
- Hardware: Probeentwurf auf Register-Transfer-Niveau soweit verfeinern, daß Schaltungstiefen wenigstens abgeschätzt werden können. Evtl. Feinentwurf (bzw. Gate Level Synthesis) der leistungsbestimmenden Kombinatorik.

Aufwandsvergleich

Die Hardware-Effizienz
Der reine Kostenvergleich genügt oftmals nicht. Vielmehr sind Kosten-Nutzen-Betrachtungen anzustellen. Hierfür liegt es zunächst nahe, Zahlenwerte im Sinne einer *Hardware-Effizienz* HE zu bilden:

$$HE = \frac{\text{Leistungsangabe}}{\text{Aufwandsangabe}}$$

Die Leistungsangabe kann beispielsweise die Ausführungszeit sein oder ein Leistungsmaß in Nutz-Bits/s, die Aufwandsangabe ist je nach Problemstellung zu wählen (Kosten, Strombedarf, Fertigungsstunden usw.).

Die Mehraufwandseffizienz
Dies ist ein Maß, mit dem verschiedene Varianten hinsichtlich ihrer Zweckmäßigkeit untereinander verglichen werden können. Typisches Beispiel: Wir haben einen ersten Entwurf (Variante 1) und entwickeln daraus etwas Leistungsfähigeres (Variante 2: mehr Speicher, schnellerer Prozessor, Übergang auf eine leistungsfähigere Technologie usw.). Die Praxis zeigt, daß beides vorkommen kann:

a) Mit vergleichsweise geringen Zusatzaufwendungen erhalten wir überproportional mehr Leistung.
b) Selbst extreme Aufwandserhöhung bewirkt nur eine vergleichsweise geringe Leistungssteigerung.

Zur Bewertung eignet sich die *Mehraufwandseffizienz* (des Übergangs von Variante 1 zu Variante 2) $ME_{1\text{-}2}$:

$$ME_{1-2} = \frac{HE_2}{HE_1}$$

Hierin sind HE_1, HE_2 die Werte der Hardware-Effizienz beider Varianten.

Der Wirkungsgrad
Diese Angabe soll das Leistungsvermögen einer bestimmten Implementierung (in Hard- oder Software) in Bezug auf eine leistungsmäßig "ideale" Implementierung bzw. auf die absoluten Leistungsgrenzen kennzeichnen.

$$\eta = \frac{P_{var}}{P_{ideal}}$$

- η: Wirkungsgrad ($0 < \eta \leq 1$),
- P_{var}: Leistungsangabe der jeweils zu untersuchenden Variante,
- P_{ideal}: Leistungsangabe der "idealen" Implementierung (= absolute Leistungsgrenze bei Ausnutzung des Standes der Technik). Wir beziehen uns hier typischerweise auf den Probeentwurf einer kompromißlos auf Leistung ausgelegten Spezialmaschine (diese muß gemäß Stand der Technik im Rahmen plausibler Kostenlimits ausführbar sein; die Kostenvorgaben des aktuellen Projekts werden aber nicht berücksichtigt).

Hinweis: Die angeführten Aufwands- und Leistungsangaben werden typischerweise auf Grundlage von "angearbeiteten" Probeentwürfen (oder -programmen) abgeschätzt. Die Praxis zeigt, daß sich Größenordnungen und Tendenzen (beim Variantenvergleich) ziemlich schnell erkennen lassen[2].

Bemühungen, die sich auch beim derzeitigen Stand der Technik noch lohnen

Wichtig ist vor allem, Anwendungslösungen in einem akzeptablen Zeitrahmen erstellen zu können. Hardwarelösungen haben bei aller Leistungsfähigkeit den Nachteil, daß sie längere Entwicklungszeiten und mehr Können erfordern: "Programmieren kann heutzutage jeder" (es fragt sich zwar immer noch, wie – aber das Programmier-Paradigma ist heutzutage wohl so allgemein verinnerlicht worden, daß man mehr brauchbare Programmierer finden dürfte als brauchbare Hardware-Entwickler). Höhere Integrationsgrade verschärfen das Problem eher noch (wieviele Menschen gibt es, die mit mehreren 100k Gattern wirklich etwas Sinnvolles anstellen können?).

Die Aufgabe der einschlägigen Forschung ist deshalb vor allem darin zu sehen, bessere programmierbare Strukturen zu schaffen, um spezielle Hardware-Entwicklungen noch weiter entbehrlich zu machen.

Solche Lösungen sollten offen und zukunftssicher sein. Dazu sollten sie auf hinreichend abstrakten Grundlagen beruhen. Dies bildet die Gewähr dafür, die Entwicklungsleistungen auch nach Jahren noch reproduzieren zu können (Ersatzteilversorgung). Niemand kann die Mathematik monopolisieren (wenigstens bis jetzt noch nicht) – und sie hängt auch nicht von veränderlichen Industriestandards ab.

Zuhandenheit auf Grundlage scheinbar offener (weil allgegenwärtiger), tatsächlich aber proprietärer Standards ist trügerische Sicherheit[3].

Die Lösung einer Entwicklungsaufgabe müßte zunächst zu einer funktionellen Spezifikation führen, die je nach Stand der Technik in eine Implementierung umgesetzt wird (Ressourcen-Algebra). Hardware- und Architektur-Plattformen sollten dies unterstützen. Derartige Plattformen sollten auf dem jeweils einfachsten Paradigma aufsetzen (virtuelle Maschinen).

Zur Ideengeschichte: K. Zuse (systematische Definition der Informationsstrukturen und Informationswandlungen vom Bit an) – APL (Bereitstellung allgemein bewährter höher aggregierter Informationsstrukturen) – ADA/VHDL (Package-Konzept, Trennung von Spezifikation und Implementierung, eine gemeinsame Sprach-Grundlage für Funktions- und Strukturbeschreibungen).

2: Einen gewissen Mindest-Arbeitsaufwand vorausgesetzt – man muß schon mehr tun als lediglich in Prospekten und Datenbüchern zu blättern ...

3: Microsoft, Intel usw. sind ganz gewöhnliche Privatfirmen, keine Gottheiten ...

Einschlägige Forschungs- und Entwicklungsvorhaben in Stichworten:

- Verkopplung zuhandener Prozessoren/Controller zu Multiprozessorkonfigurationen,
- Ergänzung zuhandener Prozessoren/Controller durch Zusatzbeschaltung (betrifft auch Prozessorkerne in ASICs und FPGAs),
- Ressourcen-Algebra als hersteller- und implementierungs-unabhängige Formalbeschreibung,
- Hardware-Multitasking,
- problemangepaßte Verarbeitungsbreiten (Mikrocontroller als Meterware),
- Hardware-Vergegenständlichung grundlegender Konzepte: Boolesche Gleichungen, State Machines, Zustands- und Ereignissteuerung,
- Einbau von Debugging- und Schutzvorkehrungen,
- Simulation der Register-Transfer-Ebene (Prinzip: Problemlösung, falls zweckmäßig, als Hardware entwerfen, aber als Software ausführen lassen),
- ungewöhnliche Elementaroperationen (relationale Operationen über Binärvektoren, Carry-Save-Arithmetik usw.).