Quantitative Prinzipien im Hardwareentwurf

1. Small is fast

Kleine Hardwareeinheiten schalten in der Regel schneller als größere. Kleine Transistoren bilden an ihren Gates kleinere Kapazitäten die Source-Drain Strecken bilden kleinere Widerstände. Daher können Umladevorgänge schneller stattfinden. Kürzere Leitungen bilden ebenfalls kleinere Kapazitäten und Widerstände. Ein Signal kann auf einer kurzen Leitung schneller übertragen werden als auf einer langen. Kleinere Einheiten können mit kleineren Ausgangsleistungen geschaltet werden, d.h. die Transistoren, die diese Treiben müssen können selbst kleiner und dadurch schneller sein.

Eine Erhöhung der Verlustleistung kann zu einer Beschleunigung führen. Man kann diese erhöhte Verlustleitung in der Regel nur bei kleinen Schaltungen in Kauf nehmen, weil sonst der gesamte Chip zu heiß wird. Dies haben wir am Beispiel der Differenzverstärker in Speicherbausteinen gesehen.

2. Make the common case fast

Wenn man eine Wechselwirkung im Entwurf erkennt, daß man einen Fall schneller machen kann auf Kosten eines anderen, so ist stets der häufigere Fall zu favorisieren. Wenn zum Beispiel zwei Zahlen zu addieren sind, so kann ein Überlauf auftreten. Dieser Fall ist aber selten. Es ist daher sinnvoll, die Addition ohne Überlauf so schnell wie möglich zu machen, sogar, wenn dadurch der Fall mit Überlauf langsamer wird.

3. Amdahls Gesetz

Ein Gesetz, das diesen Zusammenhang quantitativ fassbar macht ist **Amdahls Law:**

Angenommen, wir können eine Veränderung an einer Maschine vornehmen, die eine bestimmte Operation beschleunigt.

Def:

$$speedup = \frac{Ausführungszeit\ der\ gesamten\ Aufgabe\ ohne\ Verbesserung}{Ausführungszeit\ der\ gesamten\ Aufgabe\ mit\ Verbesserung}$$

Amdahls law gibt schnell darüber Aufschluß, wie groß der Speedup bei einer Veränderung ausfällt. Dazu ist erforderlich, daß zwei Parameter bekannt sind:

1.) Der Anteil der beschleunigbaren Operationen an der gesamten Aufgabe, genauer, der Zeitanteil, den diese Operationen einnehmen. Dieser Zeitanteil wird als **$\text{Anteil}_{\text{beschleunigt}}$** bezeichnet.
Beispiel: die Aufgabe braucht unbeschleunigt 60 Sekunden. In 20 Sekunden wird eine Operation ausgeführt, die beschleunigt werden kann. Dann ist der $\text{Anteil}_{\text{beschleunigt}}$ 60s/20s=1/3.

2.) Die Verbesserung bei der beschleunigbaren Operation, die erreicht werden kann.
$\text{Speedup}_{\text{beschleunigt}}$
Beispiel: Operation braucht unbeschleunigt 5ns, beschleunigt 2ns, $\text{Speedup}_{\text{beschleunigt}}$=2,5

$$Ausführungszeit_{unbeschleunigt} = Ausführungszeit_{unbeschleunigt} * (1 - Anteil_{beschleunigt} + Anteil_{beschleunigt})$$

$$Ausführungszeit_{beschleunigt} = Ausführungszeit_{unbeschleunigt} * (1 - Anteil_{beschleunigt} + Anteil_{beschleunigt} / Speedup_{beschleunigt})$$

$$\frac{Ausführungszeit_{unbeschleunigt}}{Ausführungszeit_{beschleunigt}} = \frac{1}{1 - Anteil_{beschleunigt} + \frac{Anteil_{beschleunigt}}{Speedup_{beschleunigt}}}$$

Amdahls Gesetz:

$$Speedup = \frac{1}{1 - Anteil_{beschleungt} + \frac{Anteil_{beschleungt}}{Speedup_{beschleungt}}}$$

Beispiel:
Angenommen wir können eine Operation um einen Faktor 10 beschleunigen, die zu 40% der Ausführungszeit ausgeführt wird. Welchen Speedup erreichen wir dadurch für die gesamte Ausführung?

$\mathbf{Anteil_{beschleunigt} = 0{,}4}$

$\mathbf{Speedup_{beschleunigt} = 10}$

$\mathbf{Speedup_{gesamt} = 1/ (0{,}6 + 0{,}4/10) = 1/0{,}64 = 1{,}56}$

Amdahls Law ist ein hervorragendes Werkzeug, mit dem unterschiedliche Designalternativen gegeneinander bewertet werden können.

Beispiel:
Angenommen SQRT ist verantwortlich für 20% der Ausführungszeit eines Benchmarks auf einer Maschine. Ein Vorschlag ist, SQRT durch Spezialhardware um einen Faktor 10 zu beschleunigen, ein anderer, alle FP-Operationen doppelt so schnell zu machen. FP-Operationen machen dabei 50% der gesamten Ausführungszeit aus. Welches ist die bessere Wahl?

Amdahls Law ist ein hervorragendes Werkzeug, mit dem unterschiedliche Designalternativen gegeneinander bewertet werden können.

Beispiel:
Angenommen SQRT ist verantwortlich für 20% der Ausführungszeit eines Benchmarks auf einer Maschine. Ein Vorschlag ist, SQRT durch Spezialhardware um einen Faktor 10 zu beschleunigen, ein anderer, alle FP-Operationen doppelt so schnell zu machen. FP-Operationen machen dabei 50% der gesamten Ausführungszeit aus. Welches ist die bessere Wahl?

$\mathbf{Anteil_{beschleunigt} = 0,2}$
$\mathbf{Speedup_{beschleunigt} = 10}$
$\mathbf{Speedup_{gesamt} = 1/ (0,8 + 0,2/10) = 1/0,82 = 1,22}$

$\mathbf{Anteil_{beschleunigt} = 0,5}$
$\mathbf{Speedup_{beschleunigt} = 2}$
$\mathbf{Speedup_{gesamt} = 1/ (0,5 + 0,5/2) = 1/0,75 = 1,33}$

Der zweite Vorschlag ist besser.

4. Die CPU-Performance-Gleichung

Jeder Computer hat einen Takt mit einer festen Taktfrequenz, z.B. 500 MHz. Der Takt teilt die Zeit in lauter kleine Abschnitte, so genannte Taktzyklen. Die Länge der Taktzyklen ist genau der reziproke Wert der Taktfrequenz:

Beispiel: f=500MHz => Zykluszeit = $1/(500 * 10^6 * 1/s) = 1/5 * 10^{-8}$ s = $2\ 10^{-9}$ s =2 ns

Die CPU-Zeit eines Programms kann daher auf zwei Weisen angegeben werden:

CPU-Zeit = Anzahl der Taktzyklen für das Programm * Zykluszeit

oder

CPU-Zeit = Anzahl der Taktzyklen für das Programm / Taktfrequenz

Andererseits können wir auch die Anzahl der Instruktionen (**IC=instruction count**) zählen, die in dem Programm ausgeführt werden.

Wenn wir die Gesamtanzahl der Taktzyklen des Programms kennen, dann können wir den Durchschnittswert für die Anzahl der Taktzyklen pro Befehl **CPI, clocks per instruction** ermitteln

CPI = Anzahl der Taktzyklen des Programms / IC

Der CPI Wert ist für die quantitative Analyse ein ausgesprochen wichtiger Parameter, der Einsicht in die Effizienz unterschiedlicher Instruktionssätze gibt. Wir werden ihn häufig benutzen.

Durch Einsetzen dieser letzten Gleichung in die CPU-Zeit Gleichung erhalten wir

CPU-Zeit = IC * CPI * Zykluszeit = IC * CPI / Taktfrequenz

Ausführlicher hingeschreiben bedeutet diese erste Formel:

$$CPU\ Zeit = Anzahl\ Instruktionen\ für\ Programm \times \frac{Taktzyklen}{Instruktion} \times \frac{Sekunden}{Taktzyklus} = Sekunden\ für\ Programm$$

Die CPU-Zeit hängt von diesen drei Parametern gleichermaßen ab. Eine Verbesserung in einem von ihnen um 10% verbessert die Performance um 10%.

Beispiel:

Ein Programm auf einem 3 GHz-Rechner führt im Laufe seiner Ausführung $3{,}42 * 10^{11}$ Befehle aus. Die CPI für dieses Programm ist 1,22. Wie lang ist die Ausführungszeit?

Aufgabe:

Ein Programm auf einem 3 GHz-Rechner führt im Laufe seiner Ausführung $3{,}42 * 10^{11}$ Befehle aus. Die CPI für dieses Programm ist 1,22. Wie lang ist die Ausführungszeit?

Lösung:

Taktzykluszeit:
3 GHz bedeutet $3 * 10^9$ Hz = $3 * 10^9 * 1/s$;
Also Taktzykluszeit = $1/(3 * 10^9)$ s = $0{,}333 * 10^{-9}$ s = 0,333 ns

CPU-Zeit = $3{,}42 * 10^{11} * 1{,}22 * 0{,}333 * 10^{-9}$ s = 139,08 s

Unglücklicherweise ist es schwierig, sie in Isolation voneinander zu sehen, denn die Technologien, die für jede dieser Größen verantwortlich sind, sind ebenfalls abhängig voneinander.

Zykluszeit: Hardware Technologie und Organisation
CPI: Organisation und Befehlssatz-Architektur
IC: Befehlssatz-Architektur und Compiler Technologie

Glücklicherweise gibt es oft die Möglichkeit, einen der Parameter stark zu verbessern mit nur moderater und kalkulierbarer Verschlechterung der beiden anderen.

Oft ist es sinnvoll, die Anzahl der Taktzyklen für ein Programm detaillierter anzugeben: Man ermittelt dann die CPI für einzelne Instruktionenklassen anstelle für den gesamten Instruktionssatz.

$$Anzahl\ der\ Taktzyklen = \sum_{i=1}^{n} CPI_i \times IC_i$$

Dadurch wird die CPU-Performance-Gleichung zu

$$CPU\text{-}Zeit = \sum (CPI_i * IC_i) * Zykluszeit$$

Und die CPI für den gesamten Befehlssatz

$$CPI = \frac{\sum_{i=1}^{n} CPI_i \times IC_i}{IC} = \sum_{i=1}^{n} CPI_i \times \frac{IC_i}{IC}$$

Beispiel:
Anteil der Gleitkommaoperationen 25%
Durchschnittliche CPI der Gleitkommaoperationen 4
Durchschnittliche CPI der anderen Operationen 1,33
Anteil DIV 2%
CPI von DIV 20

Zwei Design-Alternativen:

1. **Senken der CPI aller FP-Operationen auf 3 oder**
2. **Senken der CPI von DIV auf 2.**

Was ist besser?

Beispiel:
Anteil der Gleitkommaoperationen 25%
Durchschnittliche CPI der Gleitkommaoperationen 4
Durchschnittliche CPI der anderen Operationen 1,33
Anteil DIV 2%
CPI von DIV 20

Zwei Design-Alternativen: Senken der CPI aller FP-Operationen auf 3 oder Senken der CPI von DIV auf 2. Was ist besser?

CPI = 75% * 1,33 + 25% * 4 = 2
CPI1 = 75%*1,33 + 25% * 3 = 1,75

CPI = 98%* $CPI_{\text{alles außer DIV}}$+2% * 20
CPI2 = 98%* $CPI_{\text{alles außer DIV}}$+2% * 2
Gleichungen voneinander abziehen:
CPI - CPI2 = 2%*(20-2) =0,36
CPI2 = CPI - 0,36 = 2 - 0,36 = 1,64
Das ist besser als 1,75.

$$speedup = \frac{CPU\ Zeit_{alt}}{CPU\ Zeit_{neu}} = \frac{IC \times Zykluszeit \times CPI}{IC \times Zykluszeit \times CPI2} = \frac{CPI}{CPI2} = \frac{2,0}{1,64} = 1,22$$

Häufig ist es schwierig, den Anteil der Zeit zu ermitteln, den eine Menge von Instruktionen in einem Programm benötigt. Dieses wäre aber notwendig für die Anwendung von Amdahls Gesetz. Daher ist die CPU-Performance-Gleichung in solchen Fällen extrem hilfreich, da man hier nur die CPI der Instruktionsmenge zu kennen braucht, und dann die Aufrufe der Instruktionen zählen kann, um so zum IC zu kommen.

Ein weiteres Beispiel soll uns zusätzliche Erfahrung im Umgang mit der CPU-Performance-Gleichung bringen:

Eine Maschine A benutzt zwei aufeinanderfolgende Befehle für einen bedingten Sprung:
F := X-Y / Compare
if F<0 then branch Label / Conditional Branch
Die Maschine B macht dasselbe in einem Befehl:
If X<Y then branch Label / Conditional Branch
Weil der Cond. Branch-Befehl in B komplexer ist, kann A mit einer Taktfrequenz betrieben werden, die um den Faktor 1,25 höher ist als die von B.
20% der Befehle bei A sind Cond. Branch-Befehle, (daher weitere 20% Compares)
Die CPI in beiden Maschinen für Cond. Branch-Befehle ist 2, für alle anderen Befehle 1.
Welche Maschine ist schneller?

CPI_A= 0,2 * 2 + 0,8 *1 = 1,2
Der Anteil von 100 Befehlen bei A ist 20 Branch, 20 Compare, 60 andere.
Da bei B die Compare-Befehle wegfallen ist der Anteil von 80 Befehlen bei B 20 Branch und 60 andere.
Also A: 20% Branch, 80% andere
B: 20/80=25% Branch, 75% andere
CPI_B = 0,25 * 2 + 0,75 * 1 = 1,25
der IC_B ist geringer als der von A, weil die Compares wegfallen. Also $IC_B = IC_A$ * 0,8
Die $Zykluszeit_B = 1,25 * Zykluszeit_A$

Es gilt also
$CPU\text{-}Zeit_A = IC_A * CPI_A * Zykluszeit_A = IC_A * 1,2 * Zykluszeit_A$
$CPU\text{-}Zeit_B = IC_B * CPI_B * Zykluszeit_B = 0,8 * IC_A * 1,25 * 1,25 * Zykluszeit_A$
$= IC_A * 1,25 * Zykluszeit_A$

Also A ist schneller

5. Lokalität

Während Amdahls Law auf jedes System zutrifft, gibt es auch quantitative Eigenschaften, die mit Programmen zusammenhängen:

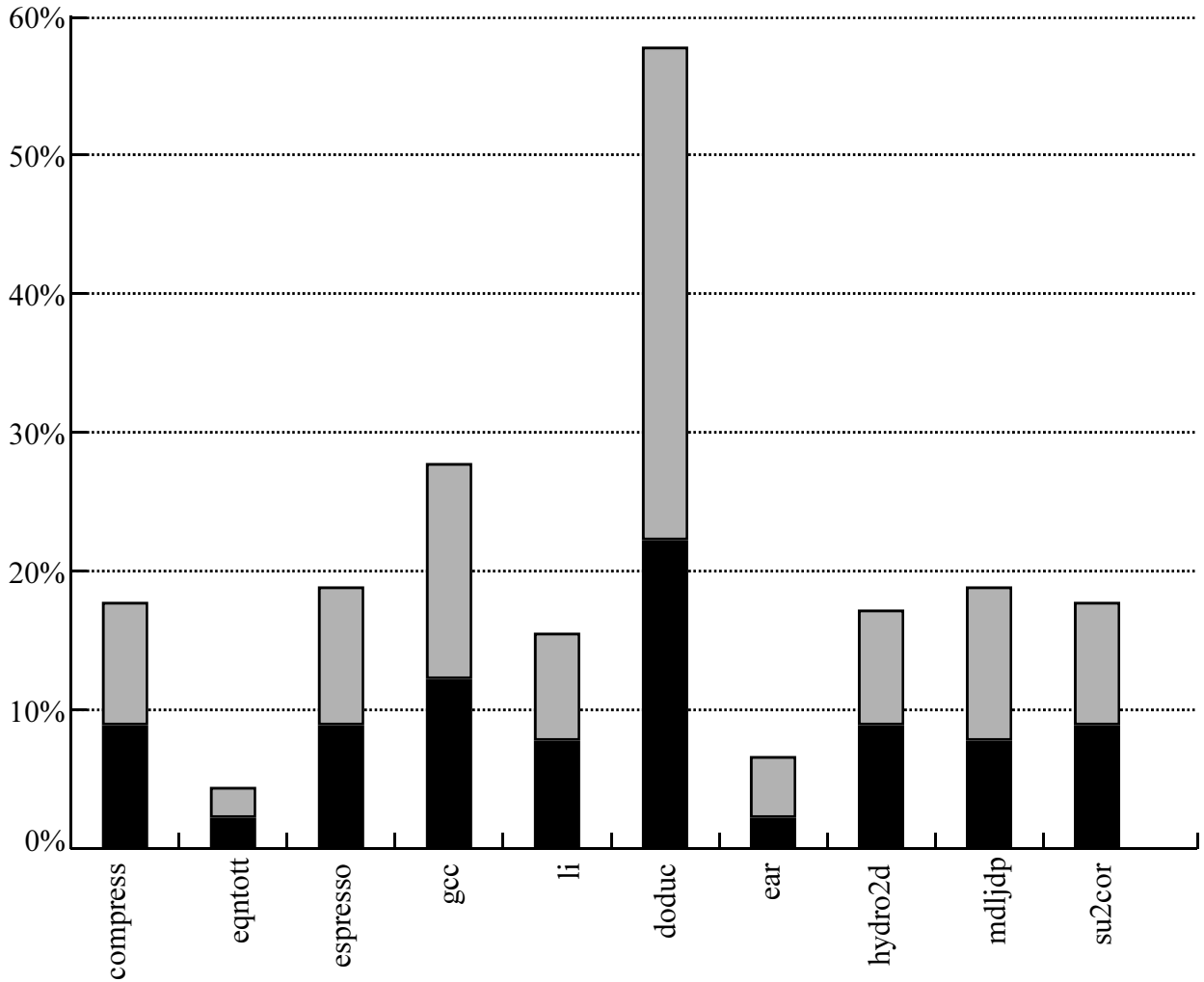
Die wichtigste davon ist **Lokalität**.

Programme haben die Eigenschaft, Daten und Befehle wieder zu benutzen, die gerade vor kurzer Zeit benutzt worden sind. Dies bezeichnet man als **zeitliche Lokalität**.

Eine Daumenregel, bekannt als 90-10-Regel besagt:

10% des Programmcodes eines Programms sind für 90% der Ausführungszeit verantwortlich.

Was nützt uns dieses Wissen? In Sinne von **make the common case fast** können wir dafür sorgen, daß wir auf die Daten und auf den Programmcode, die bzw. der gerade benutzt worden sind bzw. ist, schnell wieder zugreifen können. Wir können also Aussagen über die nahe Zukunft basieren auf Beobachtungen aus der nahen Vergangenheit.
Ein Beispiel für zeitliche Lokalität liefert das folgende Diagramm, das für Programme aus dem Spec-Benchmark zeigt, welcher Anteil an Programmcode für 80% bzw 90% der Rechenzeit verantwortlich ist.


60%
50%
40%
30%
20%
10%
0%
compress
eqntott
espresso
gcc
li
doduc
ear
hydro2d
mdljdp
su2cor
SPEC benchmark

Neben der zeitlichen Lokalität gibt es auch noch die **räumliche Lokalität**.
Diese besagt, daß Daten, auf die neu zugegriffen wird häufig nahe bei Daten liegen, auf die gerade vorher zugegriffen worden ist. Für Befehle eines Programms ist das sicher natürlich, aber auch für z.B. Elemente eines Arrays in einer Verarbeitung.

Im nächsten Abschnitt werden wir uns die zeitliche und räumliche Lokalität zunutze machen.

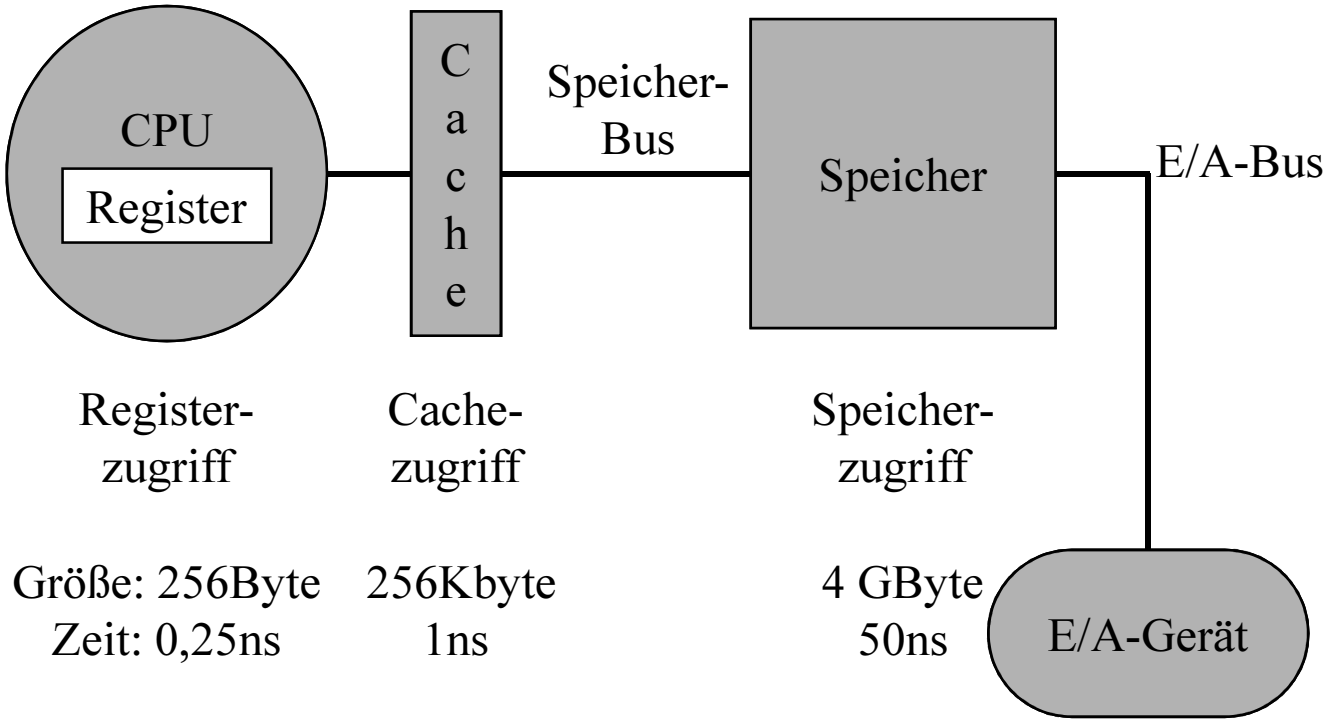
6. Anwendungsbeispiel: Das Konzept der Speicherhierarchie

Smaller is faster, daher sind kleine Speicher schneller (und kosten mehr pro Byte).

Vergrößerung von Speichern und schnellerer Zugriff sind aber Schlüsselfunktionen in der Beschleunigung von Rechnern. Daher werden wir uns das, was wir bisher gelernt haben zunutze machen, um das Prinzip der Speicherhierarchie zu verstehen.

Lokalität => kürzlich benutztes Datum wird bald wieder gebraucht. Making the common case fast bedeutet, daß wir den Zugriff hierauf beschleunigen müssen.
Smaller is faster bedeutet, daß es sinnvoll ist diese Daten in einem kleinen Speicher nahe der CPU zu halten, während die ansteigend größeren (und langsameren Speicher) immer weiter weg von der CPU anzuordnen sind.
Dies nennt man eine **Speicher-Hierarchie.**

Das folgende Bild zeigt eine typische Speicherhierarchie.


CPU
Register
C
a
c
h
e
Speicher-
Bus
Speicher
E/A-Bus
Register-
zugriff
Cache-
zugriff
Speicher-
zugriff
Größe: 256Byte
Zeit: 0,25ns
256Kbyte
1ns
4 GByte
50ns
E/A-Gerät
Platten-
speicher-
zugriff
600 GByte
5 ms

Def: Ein Cache ist ein kleiner, schneller Speicher, der räumlich nahe der CPU angeordnet ist.

Wenn auf den Cache zurgegriffen wird, und das gesuchte Datum wird gefunden, so bezeichnen wir diesen Zugriff als **Cache hit**. Wird das Datum nicht im Cache vorgefunden, so handelt es sich um einen **Cache miss**. Wenn Daten in den Cache übertragen werden, so geschieht dies in **Blöcken** fester Größe.
Zeitliche Lokalität lehrt uns, das das eben benutzte Datum sinnvollerweise im Cache aufbewahrt werden soll. Aber räumliche Lokalität sagt, daß mit großer Wahrscheinlichkeit andere Daten aus demselben Block bald zugegriffen werden.

Die Zeit, die ein cache miss braucht, ist bestimmt durch die Zeit, die wir brauchen, um einen Block vom Hauptspeicher in den Cache zu kopieren. Diese hängt ab von der Bandbreite und der Zugriffszeit des Hauptspeichers. In der Regel wartet die CPU, bis der Block übertragen ist. Die Zeit, die die CPU wartet wird als **cache miss penalty** (Strafe) bezeichnet.

Wenn der Computer einen virtuellen Speicher hat, brauchen nicht alle Daten zu jedem Zeitpunkt im Hauptspeicher gehalten zu werden, sondern sie können auf die Platte ausgelagert werden. Wiederum ist der Adressbereich in Blöcke fester Länge unterteilt, die in diesem Falle **Pages** genannt werden. Jede Page ist zu jedem Zeitpunkt entweder im Hauptspeicher oder auf der Platte.
Wenn der Computer einen virtuellen Speicher hat, so benimmt sich dieser zum Hauptspeicher so wie der Hauptspeicher zum Cache. Entsprechend dem Cache miss führt ein Zugriff auf ein Element, das auf der Platte ist zu einem **Page fault**. Dadurch wird ein Einlesen der gesamten Seite von der Platte in den Hauptspeicher initialisiert. In dieser Zeit ist es nicht sinnvoll, die CPU leer zu lassen, da das Einlesen der Seite zu lange dauert. Stattdessen wird die CPU auf eine andere Task geschaltet, während auf einen Page fault reagiert wird.
Die folgende Tabelle zeigt die typischen Größen und Zugriffszeiten jeder Ebene in der Speicherhierarchie eine Maschine im Bereich Desktop PC bis high end Server.

Speicherhierarchie

Level	1	2	3	4
Bezeichnung	Register	Cache	Hauptspeicher	Platte
Typische Größe	<1KB	<16KB	<8GB	>10 GB
Implementierung	Custom CMOS mehrere Ports	On-chip oder off-chip CMOS SRAM	CMOS SDRAM	Magnetplatte
Zugriffszeit	0,25-1ns	0,5-5ns	20-400ns	1-10ms
Bandbreite (MB/s)	4000-32000	800-5000	400-2000	4-32
Verwaltet von	Compiler	Hardware	Betriebssystem	Betriebssystem
Backup Medium	Cache	Hauptspeicher	Platte	Magnetband

Wegen der Lokalität und wegen Smaller is faster können Speicherhierarchien signifikant die Performance erhöhen:

Beispiel:
Cache: 10 mal so schnell wie Hauptspeicher. Annahme, der Cache kann in 90% der Ausführungszeit benutzt werden. Wieviel gewinnen wir an Performance?

Wegen der Lokalität und wegen Smaller is faster können Speicherhierarchien signifikant die Performance erhöhen:

Beispiel:
Cache: 10 mal so schnell wie Hauptspeicher. Annahme, der Cache kann in 90% der Ausführungszeit benutzt werden. Wieviel gewinnen wir an Performance?

Amdahls law:

speedup = 1/(1-0,9+0,9/10) = 1/0,19 = 5,3

Mit Cache steigern wir die Performance um einen Faktor 5,3.

In der Regel können wir solche Aussagen wie *der Cache kann in 90% der Ausführungszeit benutzt werden* nicht treffen. Deshalb müssen wir die quantitative Analyse von Speicherhierarchien auf die CPU-Performance-Gleichung basieren. Dazu wird die Anzahl der Taktzyklen, die die CPU auf Hauptspeicherzugriffe warten muß mit in die Gleichung einbezogen:

CPU-Zeit = (CPU-Taktzyklen + Memory-stall-Zyklen)* Zykluslänge

Die Gleichung setzt voraus, daß die Cache Hits in den CPU-Taktzyklen enthalten sind, und daß die CPU steht bei einem Cache Miss. Die Anzahl der Memory-stall-Zyklen hängt ab von der Anzahl der Cache Misses und den Kosten pro Cache Miss der Miss-penalty.

Memory stall Zyklen = Anzahl Misses * Miss-penalty Zyklen

Wenn man den durchschnittlichen Wert der Misses pro Instruktion kennt, kann man diese Formel umstellen zu

Memory stall Zyklen = IC * Misses pro Instruktion * Miss-penalty Zyklen

Wenn man die durchschnittliche Rate der Misses pro Speicherzugriff kennt, kann man diese Formel umstellen zu

Memory stall Zyklen = IC * Speicherzugriffe pro Instruktion * Missrate * Miss-penalty Zyklen