Proseminar Mikroprozessoren

Performance, Benchmarks

Norman Hendrich
Universität Hamburg, Fachbereich Informatik
Vogt-Kölln-Str. 30, D 22527 Hamburg, F314

`tech-www.informatik.uni-hamburg.de/lehre/ss2001/ps-mikroprozessoren`

Performance, Benchmarks

Performance:

- Messung von "Performance"
- Ausführungszeit vs. Durchsatz
- CPI, MIPS, MFLOPS
- Anwendungsbenchmarks

Benchmarks und -ergebnisse:

- low-level Benchmarks
- Benchmark-Suiten: BAPCO, SPEC
- Spiele, Grafikbenchmarks

- Analyse für Alpha-21164 und PentiumPro

Bewertung: Kriterien ?!

Airplane	passenger capacity	range miles	speed m.p.h.	throughput #p x m.p.h.
Boing 777	375	4630	610	228.750
Boing 747	470	4150	610	286.700
BAC Concorde	132	4000	1350	178.200
DC 8-50	146	8720	544	79.424

entsprechend für Computer:

- Gesamtzeit für Abarbeiten eines Programms
- aber für welches Programm? Office, 3D, Video, Mailserver, ...
- Durchsatz: Anzahl verarbeiteter Programme pro Zeit
- weitere Kriterien, z.B. Stromverbrauch, Gewicht, Platzbedarf, ...

Performance:

- Antwortzeit ("wall clock time", "response time", "execution time"):
  Gesamtzeit zwischen Programmstart und -ende, inkl. I/O

- Ausführungszeit (reine CPU-Zeit)

  user-time — CPU-Zeit für Benutzerprogramm
  system-time — CPU-Zeit für OS-Aktivitäten
  `Unix: time make` — `7.950u 2.390s 0:22.98 44.9%`

- Durchsatz — Anzahl bearbeiteter Programme / Zeit

- $\text{performance} = \frac{1}{\text{execution time}}$

- $\text{speedup} = \frac{\text{performance x}}{\text{performance y}} = \frac{\text{execution time y}}{\text{execution time x}}$

Takt

- fast alle aktuellen Rechner arbeiten "synchron"
- zentraler Taktgeber (clock) mit fester Rate
- z.B. Taktrate 1 GHz, entsprechend Taktperiode 1 nsec.

=>	Ausführungszeit	=	benötigte Taktzyklen	x /	Taktperiode Taktrate

=> bessere Performance durch

- höhere Taktrate (VLSI-Technik, einfache Befehle, ...)
- weniger Taktzyklen (besserer Compiler, komplexe Befehle)

- Kompromiß erforderlich (!)

CPI: Clocks per Instruction

- Prozessor führt Befehle ("Instruktionen") aus
- Zusammenhang zwischen Instruktionen und Taktzyklen ?!

CPI := mittlere Anzahl von Takten zur Ausführung eines Befehls
:= abhängig von Rechnerarchitektur und Implementation

=> Ausführungszeit für ein Programm:

Taktzyklen = Gesamtzahl der Befehle x CPI

CPU-Zeit = Gesamtzahl der Befehle x CPI x Taktperiode

$$= \frac{\text{Gesamtzahl der Befehle} \times \text{CPI}}{\text{Taktrate}}$$

CPI: Beispiel 1

- zwei CPUs mit gleichem Befehlssatz

  A: Takt 1 GHz CPI 2.0
  B: Takt 500 MHz CPI 1.2

- welcher Prozessor ist schneller?

  CPU-Zeit A = #Befehle x 2.0 x 1 ns = I x 2.0 ns
  CPU-Zeit B = #Befehle x 1.2 x 2 ns = I x 2.4 ns

=> Prozessor A ist 1.2 Mal schneller als Prozessor B

CPI: Beispiel 2

- oft dauern nicht alle Befehle gleich lang
- Ausführungszeit hängt dann vom "Befehls-Mix" ab

- Beispiel: CPU mit drei Klassen von Befehlen
- zwei Programme für die selbe Aufgabe

Klasse	CPI
A	1
B	2
C	3

# Befehle:	A	B	C
Programm 1	2	1	2
Programm 2	4	1	1

- Programm 1 führt 5 Befehle aus, Programm 2 benötigt 6 Befehle
- aber Programm 2 ist schneller:

  Takte 1 = (2 x 1) + (1 x 2) + (2 x 3) = 2 + 2 + 6 = 10
  Takte 2 = (4 x 1) + (1 x 2) + (1 x 2) = 4 + 2 + 3 = 9

Benchmarks

- Anwender mißt Performance mit seinen eigenen Programmen
- aber wie kann man die CPU-Zeit vorhersagen?

Benchmark-Programme:

- Programm zur Vorhersage / Messung von Performance

- möglichst einfach, portabel, billig, ...
- möglichst kurze Laufzeiten
- aber: oft leicht auszutricksen, z.B. spezielle Compiler-Optimierung

- reale Applikationen liefern bessere Werte
- aber bei längeren Laufzeiten / höheren Kosten

Benchmarks: Mittelung

- Zusammenrechnen mehrerer Einzel-Benchmarks?!

arithmetische Mittelung = SUM(ratio_i)
geometrische Mittelung = SQRT(PRODUKT(ratio_i))
gewichtete Mittelung

			normalized to A			to B
	Zeit A	Zeit B	A	B	A	B
Programm 1	1	10	1	10.0	0.1	1
Programm 2	1000	100	1	0.1	10.0	1
arithmetisch	500.5	55	1	5.05	5.05	1
geometrisch	31.6	31.6	1	1	1	1

Benchmarks: Beispiele

- klassische Benchmarks: Drystone, Whetstone, Linpack, ...
- oft winzige Programme mit "kuriosen" Schleifen
- regelmäßig mit "unfairen" Optimierungen umgangen

proprietäre Benchmarks:

- Ziff-Davis Winbench, Intel CPU-Index, ...
- vor allem im Windows/PC-Bereich etabliert
- Spielebenchmarks: Quake, Unreal Tournament, 3DMark, ...

Bechmark-Suiten:

- Sammlung "echter" Anwendungen mit "echten" Daten
  BAPCO, SPEC cpu92/cpu95/cpu2000, TP-C
- Spezialbenchmarks: SPEC jvm98, viewperf, ...

MIPS, MFLOPS

- besonders einfaches Maß für Performance:

MIPS := "million instructions per second"

- ein einziger Wert, skaliert mit Taktrate

- meistens Angabe des (rein theoretischen) Maximalwerts
- ignoriert CPI: z.B. 1 MIPS bei CPI=1 oder bei CPI=8
- ineffizienter Code mit hohem MIPS-Wert möglich

MFLOPS := "million floating point operations per second"

Amdahl's Gesetz

"Speedup" durch Parallelisierung? [Gene Amdahl, 1967]

System 1: berechnet Funktion X, zeitlicher Anteil 0<F<1
System 2: Funktion X' ist schneller als X mit "speedup" SX:
SX = Zeitbedarf(X) / Zeitbedarf(X')

Amdahl's Gesetz: $S_{gesamt} = \frac{1}{(1-F) + F/SX}$

=> Optimierung lohnt nur für häufige Operationen !!
=> Beispiele:

```
SX =  10, F = 0.1,  Sgesamt = 1/(0.9 + 0.01)  = 1.09
SX =   2, F = 0.5,  Sgesamt = 1/(0.5 + 0.25)  = 1.33
SX =   2, F = 0.9,  Sgesamt = 1/(0.1 + 0.45)  = 1.82
SX = 1.1, F = 0.98, Sgesamt = 1/(0.02 + 0.89) = 1.10
```


BAPCO:

"Business Applications Performance Corporation"

- "Office Productivity"

  CorelDraw / Corel Paradox (SQL) / Netscape 4.61
  MS Word / MS Excel / MS Powerpoint
  Dragon Naturally Speaking

- "Internet Content Creation"

  Adobe Premiere / Photoshp / Avid Elastic Reality
  Bryce 4 / MS Media Encoder

- Werte relativ zu einem Referenzsystem (PII-450/128 MB/Win98)
- Gesamtwert aus geometrischem Mittel

[www.bapco.com / www.madonion.com]

x86: Performance 1999 . . .

Leistungsfähigkeit der Prozessor-Familien

Prozessor	BAPCo SYSmark'98	CPU 3DMark99[1]	PovRay 3.0 [s]	Unreal [fps][2]
	besser ▶	besser ▶	◀ besser	besser ▶
bei 400 MHz, 64 MByte, Riva-TNT Grafik				
AMD K6-2	131	5588	57	25,2
AMD K6-III	151	6309	44	26,3
Intel Celeron	147	3681	42	27,7
Intel Pentium II	162	3903	44	30,6
bei 450 MHz, 64 MByte, Riva-TNT Grafik				
AMD K6-2	137	5899	52	24,7
AMD K6-III	164	7090	39	26,6
bei 500 MHz, 128 MByte, Riva-TNT2 Grafik				
Intel Celeron	178	4479	34	37,9
Intel Pentium III	192	7599	38	42,3
AMD Athlon	212	9343	27	44,1
bei 600 MHz, 128 MByte, Riva-TNT2 Grafik[3]				
Intel Pentium III	221	9060	32	45,7
AMD Athlon	238	10915	22	47,3

[1] Futuremarks 3DMark99 Max; davon der CPU-Test, der von der Grafikkarte weitgehend unabhängig ist.
[2] Unreal 2.20, 800 x 600 Punkte, 16 Bit Farbtiefe
[3] 700-MHz-Werte stehen auf Seite 132

- Performance ~ Taktfrequenz, Architekturdifferenzen irrelevant (10%)
- K6-2 ohne L2-Cache, Celeron ohne ISSE/3Dnow!

[c't 10/99 176]

x86: Performance 2000 . . .

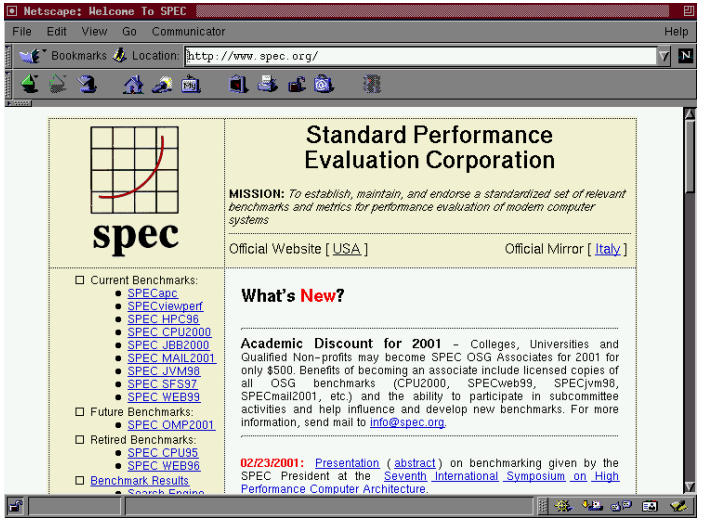
Leistungsdaten aktueller AMD- und Intel-Systeme

	Prozessor	FSB [MHz]	Speicher	Board	Windows 98 SE – BAPCo SYSmark2000 SYSmark	Internet Content Creation	Office Productivity	PovRay 3.1g PPS	Linux-Kernel sec
					besser ▶	besser ▶	besser ▶	besser ▶	◀ besser
Fliegengewicht									
AMD	K6-2/550	100	PC100-222	P5A	78	69	85	240	246
Intel	P II 450 MHz	100	PC100-222	P3B-F	93	86	98	265	252
AMD	K6-III/450	100	PC100-222	P5A	88	75	98	257	212
Intel	Celeron 500	66	PC66-222	P3B-F	94	89	97	313	245
Mittelgewicht									
Intel	FC-PGA-Celeron 600	66	PC66-222	P3B-F	112	114	111	385	206
Intel	Pentium III 600 (Katmai)	100	PC100-222	P3B-F	124	124	124	353	202
Intel	FC-PGA-Celeron 700	66	PC100-222	CUV4X	123	126	120	433	184
AMD	Athlon-600	100	PC133-333	K7V	129	128	130	468	173
AMD	Duron-650	100	PC133-333	KT133	132	134	131	515	174
AMD	Duron-700	100	PC133-333	KT133	139	141	137	556	166
Schwergewicht									
Intel	Pentium III 800	133	PC133-333	D1184	167	167	167	556	122
AMD	Athlon-800	100	PC133-222	K7V	155	159	152	614	138
Intel	Pentium III 1000	133	PC133-333	CUV4X	185	189	182	698	102
AMD	Athlon-1000 (Thunderbird)	100	PC133-333	K7V	186	187	186	800	103
Intel	Pentium III 1000 (Rambus)	133	PC800-45	VC820	197	198	197	698	101

- alle Prozessoren mit integriertem L2-Cache (außer K6-2 und Athlon)
- Performance weitgehend proportional zum Takt
- keine signifikanten Vorteile für Intel oder AMD

[c't 14/00 098]

SPEC



SPEC: Performance 03/2001

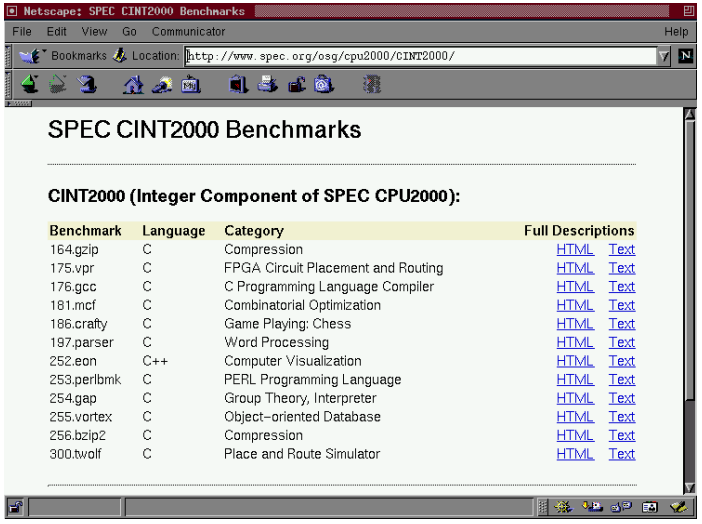
SPEC CPU2000 Benchmarks (baseline):	SPECint	SPECfp
AMD Athlon 1.2 GHz	443	387
Intel Pentium-III 1.0 GHz (VC820)	407	284
Intel Pentium-IV 1.5 GHz (VC850)	524	549
Compaq Alphaserver 833 MHz	518	590
HP 9000 j6000	417	433
Sun Blade 900 MHz	438	482

- keine offiziellen Werte für PowerPC
- alle anderen RISC weit abgeschlagen

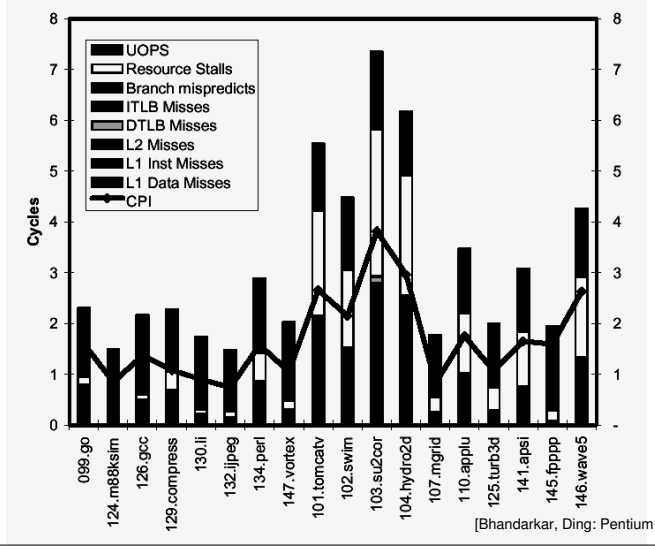
- Programme beanspruchen L1/L2-Cache + Hauptspeicher
- gleicher Speicher: sehr ähnliche Werte

[www.spec.org/osg/cpu2000, Stand 03/2001]

SPEC: CPU2000 Integer Benchmarks



SPEC: CPI vs. Stalls, Pentium Pro



[Bhandarkar, Ding: Pentium Pro characterization]

Loop: Instruction Scheduling

HW/SW-Interaktion auf Prozessoren mit Pipeline:

- Daten/Kontrollabhängigkeiten
- Wartezyklen (stalls), bis Vorgängerstufen fertig

=> sinnvolle Anordnung der Befehle notwendig
=> große Bedeutung optimierender Compiler

- Beispiel: FP-Latenzzeiten für DLX single-issue RISC aus [H&P]

instruction producing	instruction using result	latency [clocks]
FP ALU op.	FP ALU op.	3
FP ALU op.	FP STORE	2
FP LOAD	FP ALU op.	1
FP LOAD	FP STORE	0

```
ld R3, R2(0)
; wait 1
add R4, R4, R3
; wait 3
add R5, R5, R4
```


Loop: Vektor = Vektor + Skalar

- typisches Programmbeispiel: Vektor = Vektor + Skalar

```
int i; double s, x[];
...
for( i=1; i<=1000; i++) {
  x[i] = x[i] + s;
}
...
```

- nicht optimierter Code (am Beispiel DLX):

```
Loop:  LD      F0, 0(R1)    ; F0 = array element
       ADDD    F4,F0,F2     ; add scalar in F2
       SD      0(R1),F4     ; store result
       SUBI    R1,R1,8      ; decrement pointer
       BNEZ    R1, Loop     ; branch R1!=zero
```


Loop: ohne Scheduling

```
Loop:  LD      F0,0(R1)     ; F0 = array element
       ADDD    F4,F0,F2     ; add scalar in F2
       SD      0(R1),F4     ; store result
       SUBI    R1,R1,#8     ; decrement pointer
       BNEZ    R1, Loop     ; branch R1!=zero
```

- Ausführung auf der Pipeline:

```
Loop:  LD      F0, 0(R1)    ; 1  (F0 laden)
       stall                ; 2
       ADDD    F4,F0,F2     ; 3  (F0 geladen)
       stall                ; 4
       stall                ; 5
       SD      0(R1),F4     ; 6  (F4 fertig)
       SUBI    R1,R1,#8     ; 7
       BNEZ    R1, Loop     ; 8
       stall                ; 9
```

- 9 Takte / Iteration

```
int i; double s, x[];
...
for( i=1; i<=1000; i++) {
  x[i] = x[i] + s;
}
...
```


Loop: mit Scheduling

```
Loop:  LD      F0, 0(R1)    ; 1
       stall                ; 2
       ADDD    F4,F0,F2     ; 3
       stall                ; 4
       stall                ; 5
       SD      0(R1),F4     ; 6
       SUBI    R1,R1,#8     ; 7
       BNEZ    R1, Loop     ; 8
       stall                ; 9
```

- Ausnutzen des "branch delay slot": 6 Takte / Iteration

```
Loop:  LD      F0, 0(R1)    ; 1
       stall                ; 2
       ADDD    F4,F0,F2     ; 3
       SUBI    R1,R1,#8     ; 4
       BNEZ    R1, Loop     ; 5
       SD      8(R1),F4     ; 6
```

offset geändert!

```
int i; double s, x[];
...
for( i=1; i<=1000; i++) {
  x[i] = x[i] + s;
}
...
```

Loop: Unrolling

```
; solange R1 >= 3:
;
Loop:  LD      F0, 0(R1)     ; element 0
       ADDD    F4,F0,F2      ;
       SD      0(R1),F4      ;

       LD      F6,-8(R1)     ; element 1
       ADDD    F8,F6,F2
       SD      -8(R1),F8

       LD      F10,-16(R1)   ; element 2
       ADDD    F12,F10,F2
       SD      -16(R1),F12

       LD      F14,-24(R1)   ; element 3
       ADD     F16,F14,F2
       SD      -24(R1),F16

       SUBI    R1,R1,#32     ;
       BNEZ    R1, Loop      ;
```

```
int i; double s, x[];
...
for( i=1; i<=1000; i++) {
  x[i] = x[i] + s;
}
...
```

- noch kein Scheduling
- 6.8 Takte / Iteration

Loop: Diskussion

- optimierte Loop 3X schneller
- guter Compiler essentiell

aber:

- Optimierungen/Compiler nicht trivial
- maschinenspezifisch wegen Latenzen/Abhängigkeiten

- Loop-Unrolling erfordert viele Register
- erst recht für superskalare Maschinen

x86 hat zuwenig Register:

=> Compiler kann nicht optimieren

=> Register-Renaming / Tomasulo's Algorithmus

Loop: Unrolling, mit Scheduling

```
; solange R1 >= 3:
;
Loop:  LD      F0, 0(R1)     ; element 0
       LD      F6,-8(R1)     ; element 1
       LD      F10,-16(R1)   ; element 2
       LD      F14,-24(R1)   ; element 3

       ADDD    F4,F0,F2
       ADDD    F8,F6,F2
       ADDD    F12,F10,F2
       ADDD    F16,F14,F2

       SD      0(R1),F4
       SD      -8(R1),F8
       SD      -16(R1),F12
       SUBI    R1,R1,#32     ;
       BNEZ    R1, Loop      ;
       SD      8(R1),F16     ; 8-32 = -24
```

jeweils drei Takte Latenz eingehalten, aber Pipeline läuft weiter!

```
int i; double s, x[];
...
for( i=1; i<=1000; i++) {
  x[i] = x[i] + s;
}
...
```

- 3.5 Takte / Iteration
- dreimal schneller als "triviale" Version!

Loop: Register Renaming

```
; solange R1 >= 3:
;
Loop:  LD      F0, 0(R1)     ; nur F0,F2,F4 verfügbar:
       ADDD    F4,F0,F2      ;
       SD      0(R1),F4      ; => zusätzliche Abhängigkeiten

       LD      F0,-8(R1)     ;
       ADDD    F4,F0,F2      ;
       SD      -8(R1),F4     ;

       LD      F0,-16(R1)    ;
       ADDD    F4,F0,F2      ;
       SD      -16(R1),F4    ;

       LD      F0,-24(R1)    ;
       ADDD    F4,F0,F2      ;
       SD      24(R1),F4     ;
```

- x86-Compiler hat nicht genug Register zur Auswahl
- daher viele zusätzliche "Name-Dependencies"

=> "Register Renaming" zur Laufzeit im Prozessor (!)

=> ~100 Register mit "Scoreboard" zur Kontrolle der Abhängigkeiten
Beispiel Athlon: bis zu 72 Befehle aktiv ...

RISC vs. CISC: Motivation

- Rechner möglichst schnell, klein, sparsam, ..., aber billig
- sehr vielfältige Lösungen möglich
- Modeerscheinungen - z.B. "high-level instruction sets"

=> Rechnerarchitektur ist eine "Kunst"
=> gute Lösungen abhängig von HW/SW-Technologie

Ausgangsbasis für CISC: VAX, x86, 68000, ...

- Assemblerprogrammierung, schlechte Compiler
- Microcode schneller als Hauptspeicher
- Hardware für Rechenwerke vergleichsweise teuer

=> viele spezielle Maschinenbefehle
=> einmal eingeführte Befehle müssen später mitgeschleppt werden

RISC: die IBM 801

John Cocke et.al., IBM, 1975:

- warum CISC? Cache-Zugriffe genauso schnell wie Microcode...

=> Compiler-geeignete Rechnerarchitektur
=> ausschliesslich Hochsprache "PL.8": auch für OS und Treiber

=> nur wenige, reguläre Maschinenbefehle
=> aber diese schnell: Pipeline, CPI ~= 1
=> separate I/D-Caches
=> 32 Universal-Register

- 801 vs. S/370: 801 in allen Aspekten besser
- sehr guter Compiler
- wenig publiziert

RISC: RISC-I und Mips

ca 1980: 801-Nachfolgeprojekte:

- Berkeley RISC-I "reduced instruction set computer"
- Stanford MIPS "microprocessor w/o interlocked pipeline stages"

- Compiler-gerechte Architektur
- single-Chip VLSI-Implementierung

bessere Performance als 8086/68000:

• sauberer Befehlssatz	RISC
• "hardwired" Controller statt Microcode • Pipeline • viele Register, weniger Speicherzugriffe • gut optimierende Compiler • Caches, insbesondere I-Cache	auch für CISC möglich!

RISC: Designphilosophie

- minimaler, regulärer Befehlssatz
- optimale VLSI-Implementierung
- Compiler erledigt den Rest

- Berücksichtigung von Amdahl's Gesetz
- umfangreiche Performance-Simulationen (Benchmarks)

ursprüngliche RISC Entwurfsentscheidungen:

+ 32-bit Prozessor, 4 GByte Adressraum
+ 32 Universalregister (ausser RISC/SPARC)
+ 32-bit Befehlsworte, wenig Formate

- Pipeline-Abhängigkeiten (delayed branches)
- Spezialregister (MIPS mult/div)

RISC: superskalare Prozessoren

VLSI-Technologie erlaubt immer mehr Transistoren/Chip

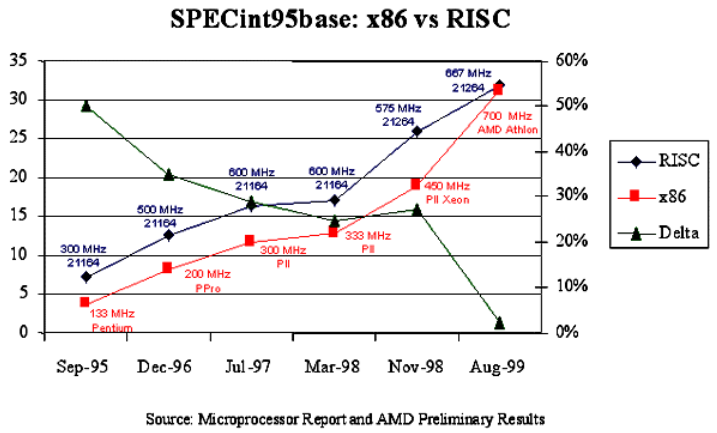
- größere Caches?
- komplexere Prozessoren?

		CPI
• klassischer CISC:	serieller Befehlszyklus	5 .. 15
• einfacher RISC:	Pipeline, 1 Befehl/Takt	~ 1
• superskalar:	mehrere Befehle/Takt	< 1

=> I-Cache muss mehrere Befehle pro Takt liefern
=> Daten/Kontrollabhängigkeiten berücksichtigen
=> Ressourcen-Konflikte, Scoreboarding
=> extreme Komplexität

=> Speicherzugriffe sind das Nadelöhr:
1 GHz, 4 Befehle/Takt, 100 ns Latenz: 400 Befehle idle

RISC vs. CISC:



Source: Microprocessor Report and AMD Preliminary Results

- vgl. aktuelle SPEC2000 Resultate, www.spec.org

superskalare Prozessoren

RISC vs. CISC für superskalare Prozessoren:	RISC	CISC
komplexe Befehlsdekodierung		•
mehrfache Funktionseinheiten	•	•
komplexes Steuerwerk (Scoreboard etc.)	•	•
out-of-order execution	•	•
große on-chip Caches	•	•
Speicherzugriffe sind das Nadelöhr	•	•

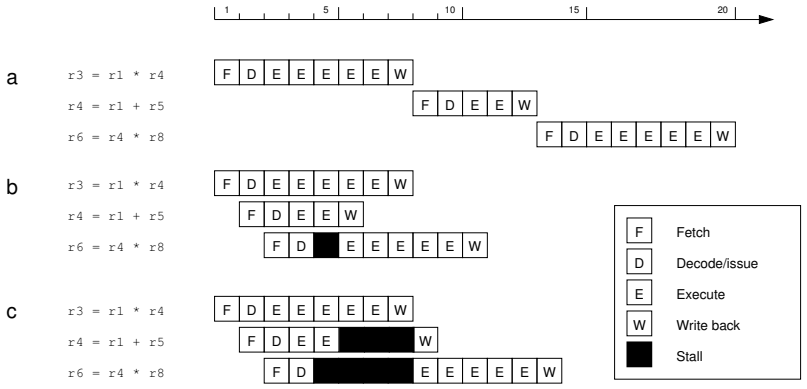
=> extreme Komplexität für RISC und CISC

- Marktbedeutung der IA-32 erlaubt große Investitionen
- bessere Chiptechnologie zuerst für x86 (Intel, AMD)
- alle x86-Prozessoren seit Pentium sind superskalar

- vgl. AMD K7 Präsentation (extern)
- K7 verwaltet bis zu 72 "instructions in flight"

Leerseite

Befehlspipeline: in order / out of order



a) serielle Befehlsbearbeitung

b) pipeline, out-of-order completion

c) in-order-completion

superskalare Prozessoren: Scoreboard

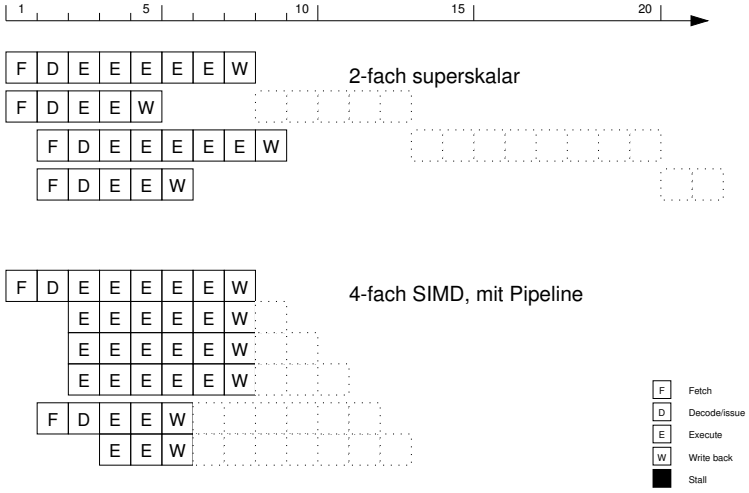
					Gelesene Register								Beschriebene Register							
Zy	#	Dekodiert	Iss	Ret	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1	1	R3=R0 * R1	1		1	1										1				
	2	R4=R0 + R2	2		2	1	1									1	1			
2	3	R5=R0 + R1	3		3	2	1									1	1	1		
	4	R6=R1 + R4			3	2	1									1	1	1		
3					3	2	1									1	1	1		
4				1	2	1	1										1	1		
				2	1	1												1		
				3																
5			4			1			1										1	
	5	R7=R1 * R2	5			2	1		1										1	1
6	6	R1=R0 – R2	–			2	1		1										1	1
7				4		1	1													1
8				5																
9			6		1		1						1							
	7	R3=R3 * R1	7		1	1	1	1					1		1					
10					1	1	1	1					1		1					
11				6		1		1							1					
12				7																
13	8	R1=R4 + R4	8						2				1							
14									2				1							
15				8																

Abb. 4.43: Operation einer superskalaren CPU mit Ausgabe und Fertigstellung von Instruktionen entsprechend ihrer Reihenfolge

Befehlsausführung
superskalar,
in-order execution
(15 Takte)

[Tanenbaum 99]

Superskalar, SIMD



superskalare Prozessoren: Scoreboard

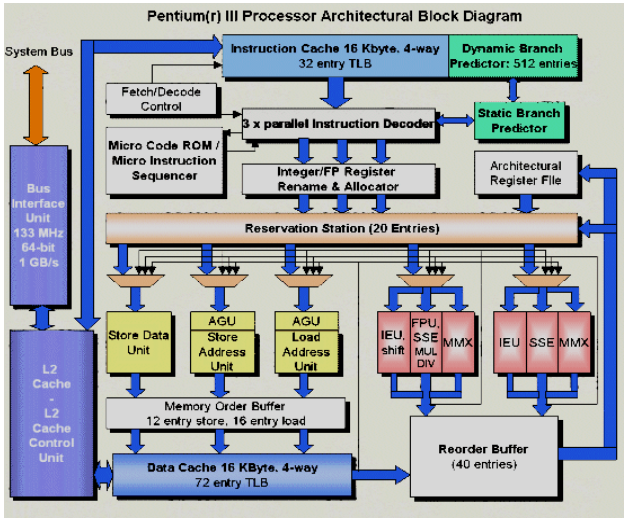
					Gelesene Register								Beschriebene Register							
Zy	#	Dekodiert	Iss	Ret	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
1	1	R3=R0 * R1	1		1	1										1				
	2	R4=R0 + R2	2		2	1	1									1	1			
2	3	R5=R0 + R1	3		3	2	1									1	1	1		
	4	R6=R1 + R4	–		3	2	1									1	1	1		
3	5	R7=R1 * R2	5		3	3	2									1	1	1		1
	6	S1=R0 – R2	6		4	3	3									1	1	1		1
				2	3	3	2									1		1		1
4			4		3	4	2		1							1		1	1	1
	7	R3=R3 * S1	–		3	4	2		1							1		1	1	1
	8	S2=R4 + R4	8		3	4	2		3							1		1	1	1
				1	2	3	2		3									1	1	1
				3	1	2	2		3										1	1
5				6		2	1		3					1					1	1
6			7			2	1	1	3					1		1			1	1
				4		1	1	1	2					1		1				1
				5				1	2					1		1				
				8				1								1				
7								1								1				
8								1								1				
9				7																

Abb. 4.44: Operation einer superskalaren CPU mit Ausgabe und Fertigstellung von Instruktionen außer der Reihe

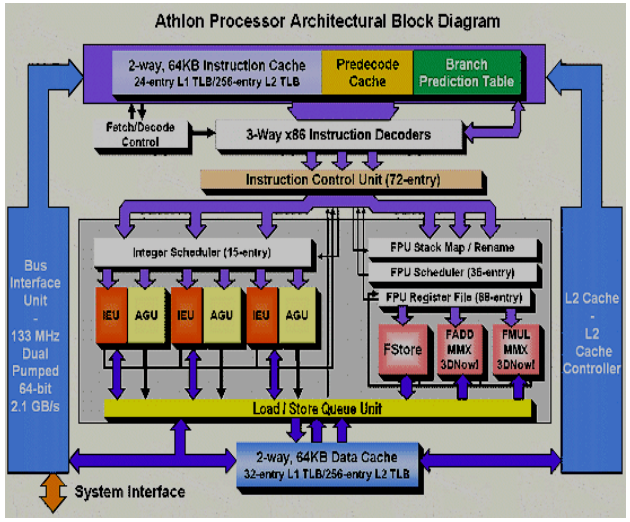
Befehlsausführung
superskalar,
out-of-order execution
(9 Takte)

[Tanenbaum 99]

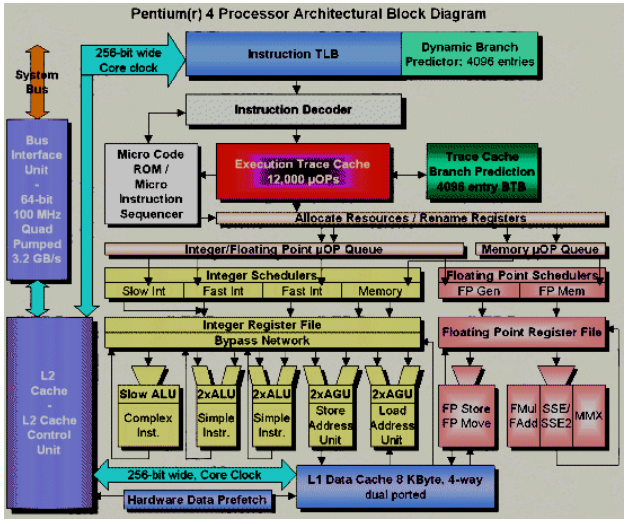
Pentium III



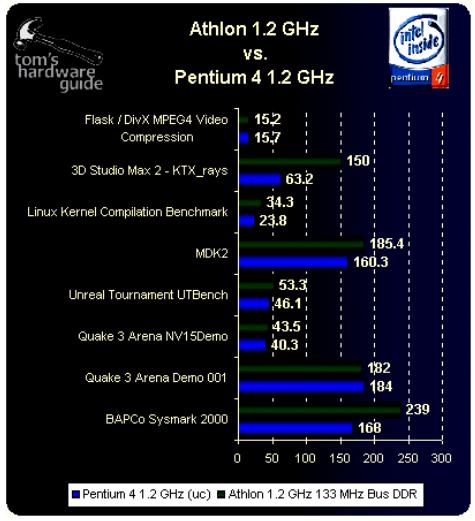
Athlon (Thunderbird)



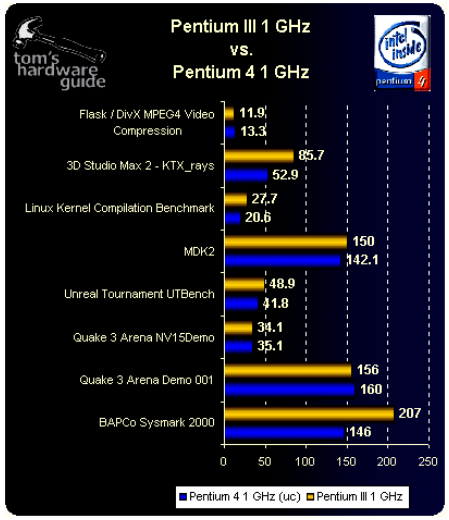
Pentium IV



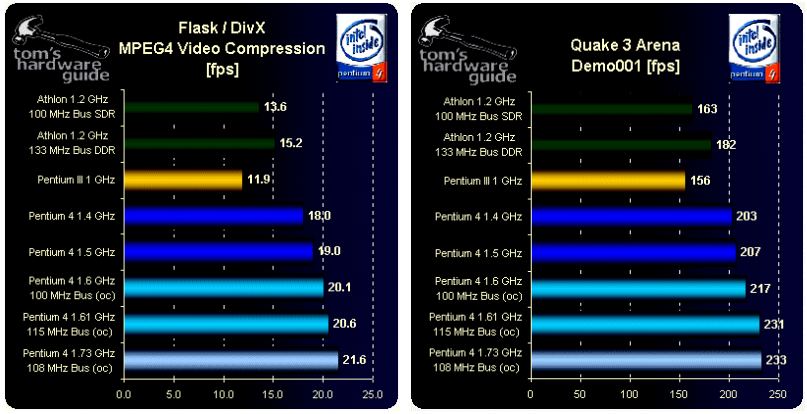
Benchmarks: Pentium IV vs. Athlon



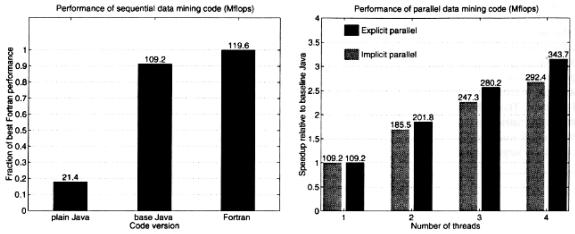
Benchmarks: Pentium IV vs. Pentium III



Benchmarks: DivX / Quake



IBM Array Package for Java



[Moreira 99]

Data mining application benchmark:

- plain Java: 18% of Fortran performance
- array package Java: 91% of Fortran performance (pure Java!)
- less bounds-checking overhead
- much better utilization of processor memory hierarchy
- good speedup on multiprocessors

IBM Array Package for Java

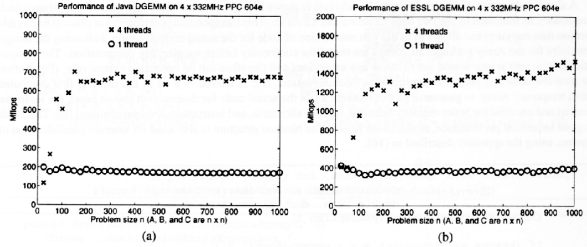


Figure 7: Performance of dgemm in (a) the Java BLAS and (b) ESSL.

[Moreira 99]

DGEMM matrix multiplication benchmark:

- array package Java reaches about 50% of Fortran performance
- Java cannot use PowerPC fma instruction (fused-multiply-add)

=> separate add/mpy halves processor performance

- good speedup on multiprocessors

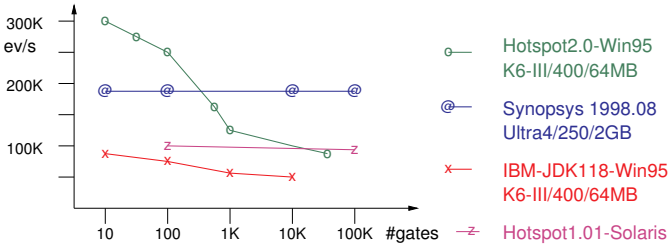
Java GC performance: discrete event simulation

discrete event simulation:

- event lifecycle: create, sort by time, process, throw away
- extreme stress for the garbage collector

=> problem: distinguish between long-lived objects and short-lived events

=> commercial simulators use their own memory management

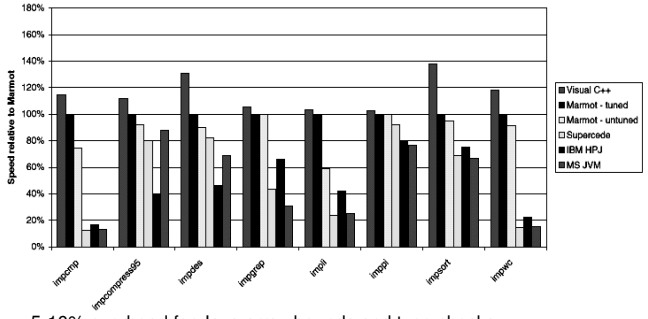


- Hotspot creates/collects up to 1M objects / sec

Microsoft Marmot

Marmot: [Fitzgerald 99]

- prototype Java compiler from Microsoft research
- compiles Java bytecode to static x86 code
- supports most of Java 1.1. class libs, aggressive optimizations
- some benchmarks with both C++ and Java version:



- 5-10% overhead for Java array-bounds and type checks

Literatur

- Hennessy & Patterson: Kapitel 2, "the role of performance"
- classical paper: Cvetanovic, Bhandarkar, ISCA 96: "performance analysis of the Alpha 21164 µP using TP and SPEC workloads"
- classical paper: Bhandarkar, Ding, "performance characterization of the PentiumPro processor", Proc. HPCA-97
- diverse Testberichte in c't und www.tomshardware.com
- www.spec.org
- www.bapco.com / www.madonion.com

Leerseite