Zentralübung Rechnerstrukturen im SS2007

Parallelismus auf Befehlsebene (korr. Fassung)

Dr. Rainer Buchty
buchty@ira.uka.de

Universität Karlsruhe (TH) – Forschungsuniversität
Institut für Technische Informatik (ITEC)
Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung

13.06.2007

Leistungssteigerung

- **Leistungssteigerung** durch
  - Taktrate
  - Überlappende Ausführung (Pipelining)
  - Intelligente Behandlung von potentiellen Konfliktsituationen (Sprungvorhersage, Eliminierung von Sprüngen durch Loop-Unrolling)
  - CPI=1 obere (unerreichbare) Schranke
- **Weitere Leistungssteigerung** durch
  - Ausnutzung von inhärentem Befehlsparallelismus (Instruction Level Parallelism, ILP)
  - Benötigt Architekturen mit mehreren, parallelen Funktionseinheiten
  - Aber: Parallelisierbarkeit begrenzt (Amdahl's Law), weitere Einflüsse durch Speichersubsystem und Verbindungsnetz
  - Grobgranularer Parallelismus (Thread & Taskebene)
- **Parallelisierung** kann erfolgen
  - Explizit durch Programmierer oder Compiler (VLIW, EPIC)
  - Mittels dedizierter Scheduling-Hardware (Superskalare Prozessoren)

Parallelismus auf Befehlsebene

- **Explizit codiert**: VLIW, EPIC
  - Direkte Kontrolle der vorhandenen Systemressourcen
  - Vom Anwender nur schwer beherrschbar
  - Compilerunterstützung notwendig
    - Loop Unrolling / Loop-level Parallelism
    - Software Pipelining
- **Dynamische Ausnutzung**: Superskalare Architekturen
  - Voraussetzung: Pipelining
  - Vervielfachung der EX-Stufe
  - Problem bei Abhängigkeiten
    - Dynamische Auflösung erforderlich
  - Erzielung maximaler Ausnutzung durch Aufbrechen der Sequentialität
    - Out-of-Order Execution
    - Erhaltung der originalen Semantik
    - Fehlerbehandlung / Interrupts

Aufgabe

Q: Geben Sie den grundsätzlichen Unterschied zwischen superskalaren und VLIW-Prozessoren an, was die Befehlsparallelisierung anbelangt.

Aufgabe

Q: Geben Sie den grundsätzlichen Unterschied zwischen superskalaren und VLIW-Prozessoren an, was die Befehlsparallelisierung anbelangt.

A: Bei VLIW- (und auch EPIC-) Prozessoren wird die Parallelisierung statisch durch den Compiler vorgenommen. In superskalaren Prozessoren wird durch entsprechende Funktionseinheiten eine dynamische Parallelisierung erzielt.

Expliziter Parallelismus

- **Beispiel:** CRYPTONITE VLIW-Architektur
  - 7 parallele Einheiten: Kontrolle, MEM1/2, ALU1/2, XOR1/2
  - Parallelismus explizit formuliert durch Füllen der 7 ,,Slots”

#	CF	MEM1	MEM2	ALU1	ALU2	XOR1	XOR2
1							
2							
3							
4							

Expliziter Parallelismus (forts.)

- **VLIW sinnvoll bei Spezialanwendungen**
  - DSP, Grafik, Netzwerk, ...
  - (Vergleichsweise) kleine Anwendungskerne
  - Typischerweise datenflußgetrieben, kaum Kontrollfluß
  - auch: Befehlssatzmodellierung (Mikroprogrammierung)
  - VLIW heute typischerweise im Hintergrund (vgl. Transmeta Crusoe) vom Anwender entkoppelt
- **VLIW problematisch im Allzweckbereich**
  - Familienbildung mit VLIW nicht machbar
  - Volle Parallelität von VLIW schwer ausnutzbar
- **Kompromiß: EPIC**
  - Beschränkung der Parallelität pro Befehlswort
  - Angabe der Einzelbefehlstypen in einem Wort
  - Familienbildung machbar, da von Parallelität auf Architekturebene entkoppelt

Dynamische Ausnutzung von Parallelismus

- Ziel: **Automatische Parallelisierung zur Laufzeit**
  - Anstoßen unabhängiger Befehle
  - Unabhängigkeit aufeinanderfolgender Befehle begrenzt
  - **Beispiel:** (a+b) * (c+d) * (e+f) * (g+h)
- **Probleme**
  - Ressourcenbeschränkungen
  - Aufeinanderfolgende Befehle eher nicht unabhängig, weiter auseinanderstehende Befehle schon
- **Dynamisierung bedeutet daher:**
  - Abkehr von strikt sequentieller Befehlsabarbeitung
  - Aufteilung des Befehlsflusses auf parallele Funktionseinheiten
  - Originale Semantik muß erhalten bleiben

Dynamische Parallelisierung

- **Parallelisierung durch Aufreißen der Sequentialität**
  - Out of Order Execution
- **Problemstellungen**
  - Erkennen von Abhängigkeiten
  - Verfügbarkeit (Einheiten und Daten)
  - Herstellung der ursprünglichen Semantik
- Zwei **grundlegende Verfahren**
  - Scoreboarding
  - Tomasulo-Algorithmus

Scoreboarding

- CDC, 1963
  - In-order issue
  - Out-of-order execute & commit
  - Behandlung der grundsätzlichen ILP-Problematik
- **Instruction Status**
  - Wo in der Pipeline?
- **Functional Unit Status**
  - Verfügbarkeit (busy/available)
  - Gewünschte Operation
  - Zielregister (F_i)
  - Quellregister (F_j, F_k)
  - Herkunft der Quellregister (Q_j, Q_k)
  - Fertigstellung via “ready”-Flags (R_j, R_k)
- **Register Result Status**
  - Welche FU schreibt welches Register?
- **Nachteile**
  - Eigenständige Funktionseinheit
  - Anpassung an Architektur notwendig (Skalierung)

Phasen des Scoreboardings

- Scoreboarding beinhaltet **4 Phasen**

1. **Zuweisung** (issue)
   - Freie Funktionseinheiten werden gesucht und potentielle WAW-Konflikte werden erkannt.
   - Sind keine Einheiten frei oder ein Konflikt wird erkannt, so wird die Abarbeitung des aktuellen Befehls angehalten (stall).

Phasen des Scoreboardings (forts.)

2. **Operanden einlesen** (read operands)
   - Die Verfügbarkeit der Quelloperanden wird überprüft.
   - Sind diese verfügbar, wird die entsprechende Funktionseinheit angewiesen, diese einzulesen und die Befehlsabarbeitung zu starten.
   - Durch diese Stufe werden RAW-Konflikte aufgelöst: Ein Operand gilt nur dann als verfügbar, wenn diesen keine bereits in Abarbeitung befindliche Instruktion beschreiben wird bzw. wenn er nicht gerade geschrieben wird.
   - Diese Definition bewirkt allerdings eine Anfälligkeit für WAW-Konflikte ($\rightarrow$ stall) und bedingt außerdem, daß WAR-Konflikte in die letzte Phase propagieren.

Phasen des Scoreboardings (forts.)

3. **Ausführung** (execution)
   - Nach erfolgter Ausführung der Operation das Scoreboard über das Ende der Ausführung informiert.
4. **Rückschreiben** (write result)
   - Auflösung von WAR-Konflikten durch “stall” und Markierung der entsprechenden Funktionseinheit als nicht verfügbar.

Tomasulo-Algorithmus

- IBM, 1966
- **Ziel:** keine speziellen Compiler benötigt
- Einsatz in Alpha, HP/PA, MIPS R10K, Pentium II, PPC604
- **Kontrolle & Puffer nicht zentralisiert** (Scoreboard) sondern in FUs verteilt → **“reservation stations”** (RS)
- Bei #RS>#Register **bessere Optimierungsmöglichkeiten als Compiler**
- Kommunikation über **gemeinsamen Datenbus** (CDB)
- **Vorteile**
  - Register nicht Flaschenhals (Renaming & Buffering)
  - Vermeidung von WAR/WAW
  - Loop-unrolling in Hardware möglich
  - nicht beschränkt auf basic blocks
- **Nachteile**
  - Komplexität
  - Viele assoziative Speicheroperationen / Geschwindigkeit
  - Flaschenhals: gemeinsamer Datenbus (CDB)

Aufgabe

Q: Scoreboarding und Tomasulo-Algorithmus sind zwei Methoden, um Out-of-Order-Execution zu ermöglichen. Worin liegt der konzeptionelle Unterschied zwischen diesen Ansätzen? Welcher potentielle Nachteil ergibt sich aus diesem Unterschied beim Tomasulo-Algorithmus?

Aufgabe

Q: Scoreboarding und Tomasulo-Algorithmus sind zwei Methoden, um Out-of-Order-Execution zu ermöglichen. Worin liegt der konzeptionelle Unterschied zwischen diesen Ansätzen? Welcher potentielle Nachteil ergibt sich aus diesem Unterschied beim Tomasulo-Algorithmus?

A: Das Scoreboard ist eine eigenständige, zentrale Einheit. Bei Tomasulo ist diese Funktionalität in den Funktionseinheiten in Form sogenannter *reservation stations* verteilt. Die Kommunikation zwischen diesen geschieht über einen eigenen Datenbus, welcher grundsätzlich einen potentiellen Flaschenhals darstellt.

Aufgabe

Q: Bei der Parallelisierung von sequentiellen Befehlsfolgen muß auf die Einhaltung der ursprünglichen Semantik geachtet werden. Was passiert bei den sogenannten Precise Interrupts und welche Einheit ist für die korrekte Auflösung dieser Unterbrechungen verantwortlich?

A: Bei Precise Interrupts muß darauf geachtet werden, daß

1. alle Resultate von Befehlen, die in der Programmreihenfolge vor dem Ereignis stehen, gültig gemacht und
2. die Resultate aller nachfolgenden Befehle verworfen werden.
3. Das Ergebnis des verursachenden Befehls wird in Abhängigkeit der Architektur oder Art der Unterbrechung gültig gemacht oder verworfen, ohne weitere Auswirkungen zu haben.

Von-Neumann-Architektur: Widerspruch?

- **Definition von-Neumann-Architektur**
  - ein Steuerwerk (“Befehlsprozessor”)
  - ein Rechenwerk (“Datenprozessor”)
  - gemeinsamer Daten- und Programmspeicher
  - Ein/Ausgabe-Einrichtung

→ gemeinsamer Daten- und Programmspeicher wesentliches Element

→ streng sequentielle Programmausführung

- **von-Neumann-Flaschenhals:**
  - strikte Sequentialität: nur skalare Operationen
  - mehrere Zugriffe durch das Rechenwerk für die Verarbeitung einer Programmanweisung

Von-Neumann-Architektur: Widerspruch? (forts.)

- **Mehrere Steuerwerke** (CMP/SMP, HT)
  - Abbildbar auf einzelne Prozessoren
- **Mehrere Rechenwerke** (Int vs. FP, Superskalarität)
  - Sequentialitätsprinzip nicht verletzt
- **Zugriffseinschränkung** durch MPU und OS
  - Geschützte Speicherbereiche (Systemverwaltung, Exklusivdaten)
  - aber: für einzelnes Programm nicht relevant
- **Speicherhierarchie** statt Speicher
  - L1/2/3-Cache, Hauptspeicher, Hintergrundspeicher
  - aber: Vereinigung durch Virtualisierung
- **Speichersegmentierung**
  - Code-, Daten-, Stacksegment
  - aber: transparent für Benutzer
- **weitere Architekturmerkmale**
  - Harvard-Architektur, Pipelining, ...
  - Gekapselt hinter ISA

daher: Von-Neumann-Prinzip weiterhin gültig

Parallelisierbarkeit: Maßzahlen und Grenzen

- **Maßzahlen**
  - Beschleunigung (Speedup): $S(n) = \frac{T(1)}{T(n)}$
  - Effizienz: $E(n) = \frac{S(n)}{n}$
  - Parallelindex: $I(n) = \frac{P(n)}{T(n)}$
- **Parallelisierung begrenzt**
  - in Hardware: Ressourcen
  - in Software: Strikt sequentielle Programmteile
- **Amdahl's Law:** Erzielbare Beschleunigung in Abhängigkeit von sequentiellem Programmanteil

$$T(n) = T(1) * (\frac{(1-a)}{n} + a)$$

Aufgabe: Leistungsfähigkeit

Gegeben sei ein Multiprozessorsystem mit 16 Prozessoren. Die Leistungssteigerung gegenüber einem Einprozessorsystem sei S(16)=8. Die Ausführungszeit auf dem Einprozessorsystem sei T(1)=80 und die Anzahl der auszuführenden Einheitsoperationen auf dem Multiprozessorsystem sei P(16)=16.

- Berechnen Sie die Effizienz E(16), Parallelindex I(16) und parallele Ausführungszeit T(16).
- Ermitteln Sie anhand von Amdahls Gesetz den Bruchteil der Programme, der nur sequentiell ausführbar ist.

Aufgabe: Leistungsfähigkeit (forts.)

Berechnen Sie die Effizienz E(16), Parallelindex I(16) und parallele Ausführungszeit T(16).

$$E(n) = \frac{S(n)}{n}, \text{ d.h. } E(16) = \frac{S(16)}{16} = \frac{8}{16} = 0.5$$

$$S(n) = \frac{T(1)}{T(n)}, \text{ d.h. } T(16) = \frac{T(1)}{S(16)} = \frac{80}{8} = 10$$

$$I(n) = \frac{P(n)}{T(n)}, \text{ d.h. } I(16) = \frac{P(16)}{T(16)} = \frac{16}{10} = 1.6$$

Aufgabe 1: Leistungsfähigkeit (forts.)

Ermitteln Sie anhand von Amdahls Gesetz den Bruchteil der Programme, die nur sequentiell ausführbar ist.

Amdahls Gesetz: $T(n) = T(1) * (\frac{(1-a)}{n} + a)$

$\rightarrow 10 = 80 * (\frac{1-a}{16} + a) = 80 * \frac{1+15a}{16} = 5 * (1 + 15a)$
$\rightarrow 15a = 1 \rightarrow a = \frac{1}{15}$

$\approx 6,7\%$ des Programmcodes sind nur sequentiell ausführbar

Aufgabe 2: Effizienz

Die Ausführungszeit einer sequentiellen Anwendung betrage $T(1)$ Sekunden. Von dieser Anwendung lassen sich 20% nicht parallelisieren und der Rest 80% werden zwischen den Prozessoren fair verteilt. “Fair” bedeutet, daß jeder Prozessor ungefähr den gleichen Anteil der zu parallelisierenden Aufgabe bearbeitet und jeder benötigt gleich viel Zeit.

Beispielsweise betrage die Ausführungszeit der parallelen Anteile der Anwendung 20% der sequentiellen Ausführungszeit, wenn vier Prozessoren sie ausführen.

- Unter der Annahme, daß die Parallelisierung keinen Aufwand verursacht. Berechnen Sie Speedup und Effizienz bei unterschiedlicher Anzahl von Prozessoren. Fügen Sie die Ergebnisse in die vorgegebenen Tabelle ein und evaluieren Sie die Skalierbarkeit der einzelnen Lösungen. Berechnen Sie den Geschwindigkeitszuwachs (Speedup).

Aufgabe 2: Effizienz (forts.)

$$T(n) = (20\% + \frac{80\%}{n}) * T(1)$$

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{0.2 + \frac{0.8}{n} * T(1)}$$

$$E(n) = \frac{S(n)}{n}$$

Aufgabe 2: Effizienz (forts.)

$$T(n) = (20\% + \frac{80\%}{n}) * T(1)$$

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{0.2 + \frac{0.8}{n} * T(1)}$$

$$E(n) = \frac{S(n)}{n}$$

	n=2	n=4	n=8	n=16	n=32
S(n)	1,67	2,5	3,33	4	4,44
E(n)	0,83	0,625	0,42	0,25	0,14

Aufgabe 2: Effizienz (forts.)

$$T(n) = (20\% + \frac{80\%}{n}) * T(1)$$
$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{0.2 + \frac{0.8}{n} * T(1)}$$
$$E(n) = \frac{S(n)}{n}$$

	n=2	n=4	n=8	n=16	n=32
S(n)	1,67	2,5	3,33	4	4,44
E(n)	0,83	0,625	0,42	0,25	0,14

Die Skalierbarkeit ist sehr schlecht. Grund dafür ist, daß ein großer Anteil vom Code nicht parallelisierbar ist. Die Ausführungszeit von diesem Anteil bestimmt einen zunehmende Anteil der gesamten Ausführungszeit mit aufsteigender Anzahl von Prozessoren.

$\lim_{n \to \infty} S(n) = \frac{1}{0.2} = 5$ und damit $\lim_{n \to \infty} E(n) = 0$

Aufgabe 2: Effizienz (forts.)

Durch jeden zusätzlichen Prozessor wird ein Aufwand von 1% der sequentiellen Ausführungszeit eingefügt. Es gilt weiterhin: Es lassen sich 20% der Aufgabenstellung nicht parallelisieren und der Rest 80% werden zwischen den Prozessoren fair verteilt.

Berechnen Sie Beschleunigung und Effizienz für 64 Prozessoren.

Aufgabe 2: Effizienz (forts.)

Durch jeden zusätzlichen Prozessor wird ein Aufwand von 1% der sequentiellen Ausführungszeit eingefügt. Es gilt weiterhin: Es lassen sich 20% der Aufgabenstellung nicht parallelisieren und der Rest 80% werden zwischen den Prozessoren fair verteilt.

Berechnen Sie Beschleunigung und Effizienz für 64 Prozessoren.

$$S(64) = \frac{T(1)}{T(64)} = \frac{T(1)}{(20\% + \frac{80\%}{64} + 1\% * 64) * T(1)} = 1{,}17$$

$$E(64) = \frac{S(64)}{64} = \frac{1{,}17}{64} = 0{,}018$$

Aufgabe 3: Beschleunigung

Ein Einprozessorsystem soll erweitert werden. Es existieren folgende, in der Anschaffung gleichteure Alternativen:

- Einsetzen eines mathematischen Koprozessors. Dieser bietet eine zweifach schnellere Ausführung der Gleitkommaarithmetik als der vorhandene Systemprozessor. Es ist allerdings keine parallele Verarbeitung, d.h. der gleichzeitige Einsatz von Haupt- und Koprozessor, möglich.
- Ausbau zu einem 2-SMP-System, d.h. die Installation eines zweiten zum vorhandenen identischen Hauptprozessor.

Das zu bearbeitende Problem ist zu 25% parallelisierbar. Der Anteil der Gleitkommaarithmetik am Gesamtprogramm beträgt 30%. Bestimmen Sie, welche der beiden Möglichkeiten unter dem Gesichtspunkt der Ausführungszeit zu bevorzugen ist.

Aufgabe 3: Beschleunigung (forts.)

Betrachten wir zunächst den einfacheren Fall, d.h. die Installation eines zweiten Prozessors. Dies macht das vorhandene System zu einem speichergekoppelten System vom UMA-Typ, da beide Prozessoren auf den gleichen, in diesem einen System installierten Speicher zugreifen.

Aufgrund der Angaben gilt somit $n = 2$ und $a = 1 - 0.25 = 0.75$. Einsetzen in die Formel für die Beschleunigung S(n) unter Berücksichtigung von Amdahls Gesetz ergibt also

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)*(\frac{1-a}{n}+a)} = \frac{n}{(1-a)+n*a}$$

also

$$S(2) = \frac{2}{0.25+2*0.75} = \frac{2}{1.75} \approx 1.14$$

Aufgabe 3: Beschleunigung (forts.)

Für das zweite – strikt sequentielle – System gilt, daß kein Parallelisierungsaufwand R(n) vorliegt, d.h. R(n)=1. Dies ist auch für das optimal parallele System der Fall, für welches $\frac{T(1)}{n} = T_{opt}$ die Laufzeit darstellt. Sei nun α der Anteil des Programms, der nicht optimiert werden kann (d.h. die Entsprechung von *a*), so läßt sich die neue Ausführungszeit T_{new} als die von α abhängige gewichtete Summe der optimierten und alten Ausführungszeit beschreiben:

$$T_{new} = (1 - \alpha) * T_{opt} + \alpha * T_{old}$$

Aufgabe 3: Beschleunigung (forts.)

Weiterhin gilt aufgrund der Angaben für die optimierbaren Programmteile $T_{old} = T_{opt} * 2$ bzw. $T_{opt} = \frac{1}{2} * T_{old}$, d.h. für die Beschleunigung ergibt sich

$$
\begin{aligned}
S' &= \frac{T_{old}}{T_{new}} = \frac{T_{old}}{(1-\alpha)*T_{opt}+\alpha*T_{old}} \\
&= \frac{T_{old}}{\frac{1-\alpha}{2}*T_{old}+\alpha*T_{old}} \\
&= \frac{2}{(1-\alpha)+2*\alpha}
\end{aligned}
$$

Somit folgt aus $\alpha = 1 - 0.3 = 0.7$ unmittelbar
$S' = \frac{2}{0.3+2*0.7} \approx 1.18 > S(2)$