Pipelining

Instruction Fetch (IF)	nimmt Befehle vom HS auf
Instruction Decode (ID)	decodiert Befehle & Register; schaut in Register & lädt Inhalte
Execute (EX)	führt Operation aus
Memory Access (MA)	macht L/S Befehle im HS
Write Back (WB)	schreibt in Register

Vorteil: Parallelisierung von Operationen
Nachteil: bei Pipelineflush müssen alle Befehle verwerfend neu geladen werden; ohne Cache immer Cachemiss

Konflikte

Write after Write	doppelte Überschreibung
Write after Read	Wert, der noch gebraucht wird, wird überschrieben
Read after Write	falscher Wert wird gelesen

Stalling

Pipeline Stalling	lässt Operationen einen Takt warten
Result Forwarding	Wert direkt weitergeben; *kein Zeitverlust, geht aber nur wenn Wert erreichbar*

Speculative Execution: raten

Befehlstypen

Abacus / MIPS	
R / R	3 Register
I / I	2 Register
S / -	1 Register
J / J	0 Register

Stalls

	ohne FW	mit FW
arithm.	2	0
L/S	2	1
Branch	3	1
Jump	3	0

Nakata

* findet min. Anzahl an Registern, die für ein Programm nötig sind
* **Knoten == Variable:**set RegNum = 1
* **Knoten unäre Operation** und Kind Label r hat: set RegNum = r
* **Knoten binäre Operation**: *
l==r: set RegNum = l+1 = r+1
* *l!=r:* set RegNum = max{l, r}
* funktioniert nicht bei geteilten Registern, z.B.: (c *~~x~~+ b) *~~x~~+ a

CPI

Befehlstyp	%	mit RAW	CPI
Sprungbefehl	20%	20%	1 bei jump; 2 bei branch
Arithmetik	60%	60%	1
Ladebefehl	20%	0.5*20%	1 wenn Wert darauf nicht gelesen, 2
MAC		0.5*20%	-

50% der Ladebefehle erzeugen einen RAW Konflikt
CPI= CPI_A*20+CPI_A*60+CPI_L*(20-10) / 100-(20-10)

Bitorientierter Cache

Datawidth	16 entspricht 2 Byte Wortgröße
MemSize	32 HS Größe
Cachesize	8 entspricht 4 Wörter
Blocksize	2 entspricht 1 Wort
SetAssoc	1 entsricht 1 Block pro Satz

direct mapping: jedes Wort genau eine Zuteilung

Byteorientierter Cache Beispiel

	tag | adr	valid
15 sti $1, $0, 0	1 in adr[0] -> 00|00	0 -> 1
16 sti $2, $0, 1	2 in adr[1] -> 00|01	0 -> 1
17 ldi $2, $0, 1	Mem[1] laden -> 00|01	1 -> hit
18 sti $3, $0, 2		
19 sti $4, $0, 3		
20 ldi $2, $0, 1		
21 sti $5, $0, 7		
22 sti $6, $0, 8		
23 sti $7, $0, 9	7 in adr[9] -> 10|01	valid = 1 && tag[9] != 00 aus 16 -> miss
24 ldi $2, $0, 1	Mem[1] laden	Cachemiss

Parsertabelle

a) ist das T in unserer FIRST-Menge drin? Dann suche Produktion, welche das T in die FIRST-Menge bringt

Parsertabelle (cont)

b) ist e in FIRST, dann schaue, ob unser T auch in der FOLLOW-Menge drin ist
Falls ja, nehme e auf

Lokalitäten

räumlich: Adressen, die nah an dem genutzten sind, werden wahrscheinlich wieder genutzt -> *Blöcke*
zeitlich: eine Adresse, auf die zugegriffen wird, wird wahrscheinlich in nächster Zeit wieder angesprochen

Zurückschreibmethoden

write through: schreibt immer durch
write back: nutzt dirtybit
schreibt zurück, wenn etwas neues eingelagert wird vom HS (oder geladen)
least recently used: *für set-oriented:* schreibt in Zeile, die am längsten nicht genutzt wurde

Graph Coloring

Spilling: nutzen den HS
1. Berechnen Read / Write-Menge & Zeilen / Blöcke
2. MayLive berechnen
3. MayUsed bestimmen
4. Zeilenweise den Schnitt nehmen
5. alle in der selben Menge haben Beziehung zueinander
6. Assembler-Code produzieren
* werden x Farben benötigt, dann ist Codierung auch mit x Registern möglich

Cacheadressierung

	Byte	Block	N_k-set
#index	$\log_2$(Cachegr.)	s_k	s_k-n_k

Cacheadressierung (cont)

#tag	$\log_2$(HSGr) - #adr	al-(s_k+b)	al-(s_k-n_k+b)
#offset	-	b	b
#gl. tags	#Zeilen im Cache	$size_k$	$size_k$/ N_k
Sätze	-	$size_k$	$size_k$/ N_k

$size_k$ = #Blöcke im Cache
s_k: $size_k = 2^{sk}$ Blöcke; B = Blockgröße in Byte;
b: $B = 2^b$; al = Bits für Adressraum
n_k: $N_k = 2^{nk}$; N_k = #Blöcke im Satz

Adressberechnung
adr(x) = x mod (Cachelines); tag(x) = x div (CL);
adr(x) = (x div 2^b) div 2^{sk}; tag(x) = (x div 2^b) mod 2^{sk}
adr(x) = (x div 2^b) div 2^{sk-nk}
tag(x) = (x div 2^b) mod 2^{sk-nk}

Blockorientierter Cache

* direct mapping
* Laden & Speichern gesamter Blöcke
+ nutzen aus, dass "nahe" Werte abhängig voneinander sind -> kein unnötiges Laden
- blöd, wenn diese unabhängig sind

Blockorientierter Cache Beispiel

15 sti $1, $0, 0
Mem[0] -> 00|0|0 = tag | in welchem Block | in welcher Zeile im Block
21 sti $5, $0, 7
Mem[7] -> 01|1|1
Cachemiss 00!=01
-> laden den Block on den HS zurück, um neuen Wert speichern zu können

Satzassoziativer Cache

* kein direct mapping
* darf sich aussuchen, in welchen Block
+ reinspeichern angenehmer, da freie Platzwahl
- suchen aufwändiger, da wir alle Blöcke im Set durchgehen müssen
* 1-way -> blockorientiert, da nur ein Block und somit keine Platzwahl

Top-Down-Parsing

Stack	input	action
$S	bbabbbb$	expand S -> bSbb
$bbSb	bbabbbb$	match
$bbS	babbbb$	expand S -> bSbb
$bbbbSb	babbbb$	match
$bbbbS	abbbb$	expand S -> A
$bbbbA	abbbb$	expand A -> aB
$bbbbBa	abbbb$	match
$bbbbB	bbbb$	expand B ->
$bbbb	bbbb$	match
$bbb	bbb$	match
$bb	bb$	match
$b	b$	match
$	$	match

S -> bSbb | A
A -> aB
B -> A |
bbabbba in Sprache?

First & Follow Beispiel

	FIRST	FOLLOW
S	b, a	**$**, FIRST(b)\e aus FOL(S):**b**
A	a	FOL(S) aus FOL(A):**$, b**
B	a,	FOL(A) aus FOL(B):**$, b**
a	a	-
b	b	-

Caches

* Cache wird bei L/S Befehlen angesprochen
* alles in binär

dirtybit: Cache konsistent zum HS?
Ja: 0; Nein: 1

validbit: ist ein Datum in der Adresse?
Ja: 1; Nein: 0

Blockorientiert: es werden nur Teile im Block überschrieben

Set-orientiert: freie Auswahl in Adresse

Cache-hit: valid = 1 & tag(i) == tag(i')

miss: valid = 0 || valid = 1 & tag(i) != tag(i')

Trefferquote: #hit/#miss

Average Acces Time: Hitrate*time(hit) + (1-Hitrate*time(miss))

Miss-Penalty: extra delay durch HS-Zugriff

First & Follow

FOLLOW:
$ aus FOL(S)
A->aBb: FIRST(b)/e aus FOL(B)
A->aB: FOL(A) aus FOL(B)
A->aBb, e aus FIRST(b): FOL(A) aus FOL(B)

Data Flow Analyse

Basic Block: * Zeile nach einem if bildet neuen Block
* if goto a bildet neuen Block an Zeile a
-> *erste Zeile im neuen Block*
-> *nur bei goto keine Beziehung zum nachfolgenden Block

Read-Menge: nehmen von allen Reads die raus, die wir im eigenen Block vorher schreiben

Write-Menge: alle geschriebenen Variablen

Variablen

* auf Klammerung achten
* am Besten Formeln aufschreiben

May Live: * backwards analysis
Live(l) = Read(l) ∪ ∪`l->l'` (Live(l') \ Write(l)), l' folgender Block

Must Live: * backwards
MLive(l) = Read(l) ∪ ∩`l->l'` (MLive(l') \ Write(l))

MayUsed: * forwards
U(l) = Read(l) ∪ ∪`l'->l`(U(l') ∪ Write(l))

MustUsed: * forwards
MustU(l) = Read(l) ∪ ∩`l'->l`(MustU(l') ∪ Write(l))

Busy: * backwards
B(l) = Read(l) ∪ ∩`l->l'` (B(l') \ Write(l))

Pipeline

CISC	register memory architecture *machinewords different length * many* complex* irregular instructions
RISC	*register register architecture* as few as posible & as regulas as possible instructions * machine words same length

shared subtrees in syntaxtrees: DAG
-> *optimaler code für DAGs ist NP-vollständig*

MSB (Most Significant Bit): linkestes Bit (höchste Potenz)

LSB (Least): rechts

vollassoziiert: wenn Cache 1 Satz (freie Speicherwahl)
n-fach assoziativ: n = Nk

Bootstrapping: Aneinanderschalten von Compilern
-> kein extra Compiler

L/S-Architektur: Befehlssatz
Daten-Speicherzugriffe nur mit L/S Befehlen
-> RISC
-> CISC hat auch ALU (arithmetic & logic unit)

Virtueller Speicher

* **TLB:** Translation-Lookaside Buffer
* **Pagetable** ist im HS