Techniky paralelního zpracování instrukcí

Břetislav Bakala

Proč je paralelní zpracování těžké?

Analogie z reálného života
1 redaktor napíše 1 článek za 2 hodiny
8 redaktorů napíše 1 článek za X hodin
Ideální paralelní zpracování: X = 15 minut

Reálné problémy
- Plánování (scheduling)
- Vyvažování zátěže (load balancing)
- Režie na komunikaci a synchronizaci (communication and synchronization overhead)

Metody paralelizace

Dosavadní metody paralelizace

- Paralelizmus na úrovni instrukčních kroků
  - Pipelining
- Paralelizmus na úrovni instrukcí
  - Superskalární zpracování

Další metody paralelizace

- Paralelizmus na úrovni dat
  - Vektorové zpracování
  - Masivně paralelní zpracování
- Paralelizmus na úrovni procesů

Optimalizace zřetězení - pipelining

- Snaha nejlépe vyvážit jednotlivé fáze zřetězení

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

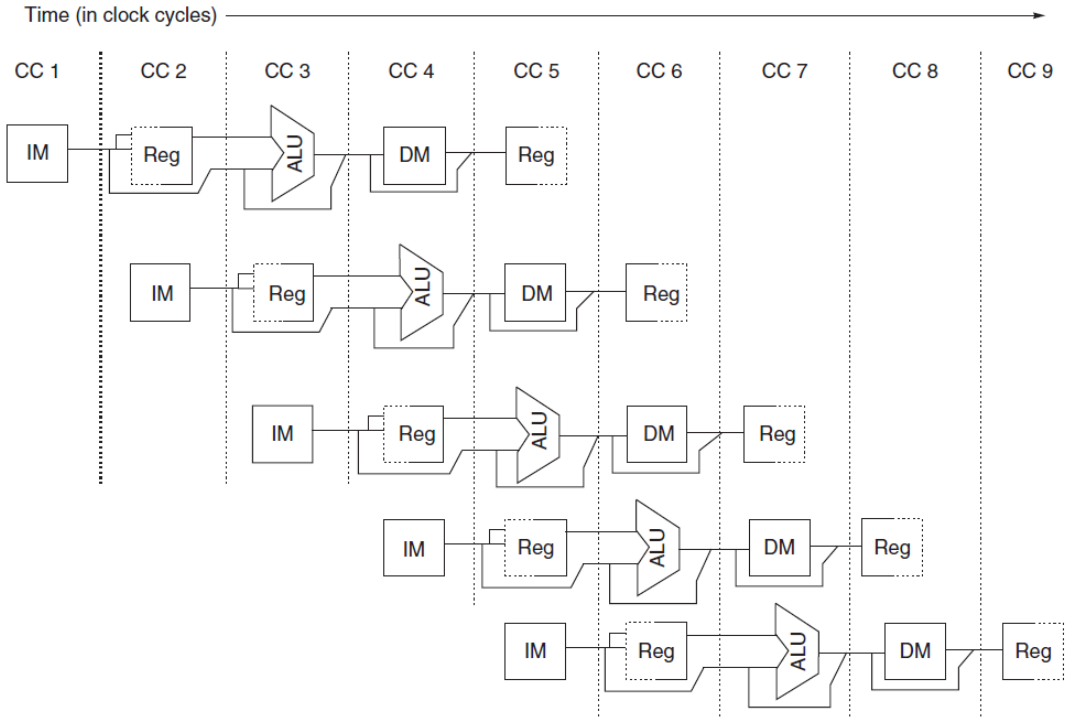
- Vykonání celočíselné operace u architektury RISC má 5 fází:

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Podmínky zřetězení

- Hlavní funkční jednotky se používají v různých cyklech
- Oddělená instrukční a datová cache
- Čtení a zápis do registru během jednoho hodinového cyklu (v první a druhé polovině) – ve fázi ID jako zdroj, ve fázi WB jako cíl
- V každém hodinovém cyklu je umožněna inkrementace stavu procesoru na novou instrukci
- Registry zabraňující rušení mezi dvěma různými instrukcemi v sousedních etapách zřetězení
- Registry se zápisem hranou - přenos dat pro danou instrukci z jedné fáze do druhé.

Podmínky zřetězení



Role registrů při zřetězení

Registry hrají klíčovou roli při provádění mezilehlých výsledků z jednoho stupně do druhého, kde zdroj a cíl nemusí být přímo sousední.
Příklady situací:

- hodnota registru, která má být uložena během instrukce uložení, je například čtena během fáze ID, ale není skutečně použita, poněvadž ve fázi MEM se prochází dvěma registry zřetězení pro dosažení datové paměti.
- výsledek instrukce ALU vypočítán během fáze EX není přímo uložen do WB, ale průchodem dvou registrů zřetězení.

Výkonové problémy zřetězení

- Zvyšuje propustnost instrukcí CPU
- Mírně zvyšuje dobu provádění každé instrukce kvůli režii při řízení pipeline
- Doba provádění jednotlivých instrukcí neklesá, což omezuje větší využití zřetězení
- Časová nerovnoměrnost jednotlivých fází zřetězení snižuje výkonnost, rychlost postupu je daná nejpomalejší fází
- Časy zápisů ustálených stavů do registrů při pipeline zpomalují zřetězení

Typy hazardů při zřetězení

- Strukturální hazard – konflikt zdrojů, hardware nepodporuje současně všechny možné kombinace instrukcí během zřetězení
- Datový hazard - instrukce závisí na výsledcích překrývající se s předchozí instrukcí ve zřetězení
- Hazard řízení zřetězení – vznikají při větvení zřetězení

Hazardy ve zřetězení mohou vést k jeho zastavení.
Předejití hazardu vyžaduje povolení některých instrukcí a zároveň zpoždění instrukcí, které by mohly vytvářet kolize.
Při zastavení instrukcí jsou zastaveny i následující instrukce.

Výkon při zastavení zřetězení I.

- Zastavení zřetězení způsobuje, že výkon zřetězení se zhorší.
- Zjištění skutečného zrychlení (speed up) je možné vyjádřit:

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$

$$= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}$$

$$= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

Výkon při zastavení zřetězení II.

- Pipelining lze podobně považovat jako snížení CPI nebo délky časového cyklu, a tím použít tradiční vyjádření pomocí CPI za předpokladu, že ideální CPI při zřetězení je vždy 1:

$$\begin{aligned}\text{CPI pipelined} &= \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction} \\ &= 1 + \text{Pipeline stall clock cycles per instruction}\end{aligned}$$

- Při zanedbání času režie zřetězení a pro stejný hodinový cyklus obou procesorů:

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}$$

Výkon při zastavení zřetězení III.

- V případě, že instrukce mají stejný počet cyklů, který se rovná počtu fází zřetězení (hloubka zřetězení - depth of the pipeline), rovná se nezřetězené CPI hloubce zřetězení.

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

- Pokud nedochází k zastavení zřetězení, odpovídá zlepšení výkonu hloubce zřetězení.

Omezení paralelního zpracování

- funkce některých programů a procesorů mohou omezit míru paralelnosti
- kritické mapování mezi strukturou programu a strukturou hardwaru je klíčové k pochopení, zda vlastnost programu skutečně omezí výkon a za jakých okolností

Hodnota CPI (počet cyklů na instrukci) pro procesor využívající zřetězení je součtem základního CPI a všech příspěvků zpomalení:

$$\text{Pipeline CPI} = \text{Ideal pipeline CPI} + \text{Structural stalls} + \text{Data hazard stalls} + \text{Control stalls}$$

Instruction-Level Parallelism (ILP)

Typickým příkladem je využití paralelismu mezi iteracemi smyčky (loop-level parallelism):

```
for (i=0; i<=999; i=i+1)
        x[i] = x[i] + y[i];
```

- Každá iterace smyčky se může překrývat. Ikdyž v rámci jedné iterace možnost překrytí není (2x načtení, součet, uložení, 2x aktualizace adres, větvení)
- Loop-level parallelism se převede na instruction-level parallelism – kompilátor rozpracuje smyčku staticky, nebo pomocí hardwaru dynamicky.

Využití SIMD při zřetězení

V předchozím příkladu paralelního zpracování smyčky je možné využít SIMD (Single Instruction Multiple Data)

- Pracuje paralelně na malém počtu datových položek (až 8)
- Využívá paralelní jednotky a hloubku zřetězení

Některé procesory mohou mít při využití SIMD namísto původních sedmi instrukcí na iteraci pouze čtyři (2x načtení vektorů, sečtení dvou vektorů, uložení výsledného vektoru). Hloubka zřetězení se redukuje na čtyři fáze. Fáze mohou být sice delší, ale vzájemně překryty.

Závislosti při zřetězení

Pokud jsou dva příkazy závislé, nejsou paralelní a musí být provedeny v pořadí, ačkoli mohou být často částečně překryty. Klíčem v obou případech je určit, zda je instrukce závislá na jiné instrukci

- Datová závislost (true data dependences)
- Závislost na umístění dat (name dependences) - dvě instrukce používají stejný registr nebo místo v paměti
- Závislost na programovém pořadí (control dependences) – zachování řídících závislostí při větvení programu

Datová závislost při zřetězení

Instrukce j je datově závislá na instrukci i, pokud nastane jedna z následujících podmínek:

- Instrukce i produkuje výsledek, který má být použit instrukcí j.
- Instrukce j je datově závislá na instrukci k a instrukce k je datově závislá na instrukci i.

Závislost na umístění dat

Rozlišujeme dva typy závislosti na umístění dat (Name Dependences) mezi instrukcí i, která předchází instrukci j:

- Antidependence – instrukce j zapisuje do registru nebo paměti, instrukce i čte ze stejného místa (nutnost zachovat pořadí)
- Output dependence - výstupní závislost nastane, když instrukce i a instrukce j zapisují do stejného registru nebo paměti. Uspořádání mezi instrukcemi musí být rezervováno tak, aby hodnota, která byla nakonec zapsána, odpovídala instrukci j.

Příkazy se závislostí na umístění dat mohou být současně spuštěny nebo znovu uspořádány, jestliže se změní název (registr nebo paměťové místo). Přejmenování může být snadněji provedeno pro operandy registru staticky – kompilátorem, dynamicky hardwarem

Datový hazard při zřetězení I.

- Pokud je hloubka zřetězení delší než je vzdálenost mezi závislými instrukcemi (měřeno v hodinových cyklech), detekuje se hazard a zastavení zřetězení.
- Může dojít v procesoru bez blokování, který závisí na plánování kompilátorů, kompilátor nemůže naplánovat závislé instrukce takovým způsobem, že se zcela překrývají.
- Překrytí během provádění změní pořadí přístupu k operandu zapojenému do závislosti
- Porušení průběhu programu, které ovlivňuje výsledek programu

Datový hazard při zřetězení II.

Možná datová rizika při závislosti na umístění dat mezi instrukcí i, která předchází instrukci j:

- RAW (čtení po zápisu) -j se pokouší číst zdroj před tím, než je zapsán, takže j nesprávně získá starou hodnotu.
- WAW (zápis po zápisu) -j se zapíše operand předtím, než je zapsán i. Zápisy se provádějí v nesprávném pořadí - výstupní závislosti. Nastává ve zřetězení, které umožní, aby instrukce i pokračovala při zastavení zřetězení.
- WAR (zápis po čtení) - j zapisuje cíl předtím, než ho i přečte, takže i nesprávně získá novou hodnotu. Vzniká při Antidependenci nebo Name Dependenci. WAR nemůže nastat v delším zřetězení, kde probíhá čtení ve fázi ID a zápisy později – ve fázi WB.

Omezení závislosti dat

- Zachováním závislosti, ale vyloučením hazardu
- Vyloučením závislosti změnou kódu

- Plánování kódu je primární metodou, která se používá k vyloučení nebezpečí bez změny závislosti
- Plánování může být provedeno jak překladačem, tak hardwarem

- Datový tok v registru - detekce závislosti je přímá (názvy registrů jsou v instrukcích pevně stanoveny)

Control Dependences

- instrukce je provedena ve správném programovém pořadí a pouze tehdy, když má být (větvení programu)

Např. závislost příkazů v části „then" příkazu if na zvolení větve

S1 je řízení závislé na p1 a S2 je řízení závislé na p2, ale ne na p1.

```
if p1 {
        S1;
};
if p2 {
        S2;
}
```

Control Dependences

Obecně platí dvě omezení řídících závislostí:

1. Instrukce, která je ovládána v závislosti na větvi, nemůže být přesunuta před zvolením větve, kde by provádění již nebylo řízeno větví. Například nemůžeme vzít instrukci z té části příkazu if a přesunout ji před příkaz if.
2. Instrukce, která není ovládána volbou větve, nemůže být přesunuta do provádění jedné z větví. Například nemůžeme přesunout příkaz původně před if do těla větve.

```
if p1 {
        S1;
};
if p2 {
        S2;
}
```

Přeuspořádání prováděných instrukcí nesmí způsobit žádné nové výjimky v programu.