02-vykonnost_pocitacu.pdf

- Jedna z možností jak definovat výkonnost počítače
  - $Performance_X = \frac{1}{Execution\ time_X}$
- $Clock\ rate = \frac{1}{Clock\ cycle\ time}$
- Pokud víme, kolik taktů bude program trvat a víme jak dlouho trvá takt (popř. víme clock rate), jsme schopni spočítat jak dlouho se program bude provádět:
  - $CPU\ execution\ time = Clock\ cycles \cdot Clock\ cycle\ time$
  - $CPU\ execution\ time = \frac{Clock\ cycles}{Clock\ rate}$
- To však obvykle nevím, víme počet instrukcí, které program má vykonat
- Avšak nevíme kolik taktů procesoru je třeba na vykonání instrukce
- **Průměrný počet taktů na vykonání instrukce**:
  - $CPI = \frac{Clock\ cycles}{Instructions}$
- **Základní vztah pro výkonnost procesoru:**
  - $CPU\ execution\ time = Instructions \cdot CPI \cdot Clock\ cycle\ time$
  - $CPU\ execution\ time = \frac{Instructions \cdot CPI}{Clock\ rate}$
- Vždy je nutné vědět všechny 3 faktory pro výpočet výkonnosti počítače, žádný není sám o sobě vypovídající
- CPI však záleží na instrukčním mixu
  - Závisí na posloupnosti instrukcí, které hodláme vykonávat
  - Při řešení kombinatorického problému, který bude mít hodně cache missů, spekulace ani predikce příliš nebudou fungovat, budeme mít **velmi nízké** CPI
- CMOS
  - Technologie, kterou se vyrábí integrované obvody
  - Minimální spotřeba v klidovém stavu
  - Ztrátový výkon
    - $P \approx \frac{1}{2} \cdot C \cdot U^2 \cdot f$
  - Za posledních 20 přibližně 1000x zvětšilo f
  - Dokud to šlo, tak jsme snižovali napětí
    - 0 — 0V
    - 1 — 1V
  - Fyzikální limity
- **Energetická zeď**
  - *Napětí není možné neomezeně snižovat*
  - *Chlazení není snadné jednoduše zlepšovat*

    - *Nové cesty ke zyšování výkonu*
        - Přechod od jednoprocesorových k víceprocesorovým architekturám
- **The Free lunch is over** (Herb Sutter)
    - Dneska se zvyšuje propustnost přidáváním jader (programy se sami nezrychlý)
    - Tlak na paralelní programování

- Mějme program, jehož celková doba běhu je 100s
    - Z toho násobení trvá 80s
- Kolikrát je třeba zrychlit násobení, aby se celkově program zrychlil 5x?
- $E_S = 80 + 20$
- $E_N = \frac{E_S}{5}$
- $E_N = \frac{80}{x} + 20$
- $20 = \frac{80}{x} + 20$ (**nemá řešení**, pouze limitním přechodem pro $x \to \infty$)
    - Využili jsme vlastně počítání dvěma způsoby (kombagra)

- Amdahlův zákon
    - "*Zvýšení výkonnosti dosažitelné nějakým zlěpšením je omezené mírou používání tohoto zlepšení*"
    - $S(x) = \frac{1}{p + \frac{1-p}{x}} \qquad 0 \leq p \leq 1,\ x \in \mathbb{N}$
- Praktické důsledky
    - "*Make the common case fast"*

04-logicke_obvody.pdf

- Sekvenční obvody jako paměť mohou být:
    - Synchronní
        - Stav se mění např. s náběžnou hranou nějakého signálu
    - Asynchronní
        - Stav se může měnit kdykoliv

05-implementace_procesoru.pdf

- Dekodér 1 z N
- Multiplexer a jak vy z dekodéru vyrobit
- Přístup do paměti pouze na bloky zarovnané na 4 byty
    - Odpovídá délce slova

- Oddělená paměť pro kód a data → Harvardská architektura
- Jednocyklová datová cesta
    - Spojení kombinačních a sekvenčních obvodů
    - Veškeré operace jsou v jednom taktu (cyklu)
    - Takt musí být dostatečně dlouhý, aby se v něm stihla i ta nejpomalejší instrukce
    - CPI = 1
    - Obecně nižší taktovací frekvence (clock cycle musí být dostatečně dlouhý)
- Řadič datové cesty (datapath controller)
    - Logický obvod generující řídící signály
    - Hodnoty signálů závisí na instrukci (jejím op codu)
    - Řadič pomocí ROM
        - Op code bude adresovat ROM, kde budou uložené správné signály k dané instrukci
        - Jak vyrobit ROM, která je rychlejší než datová cesta?
    - Řadič pomocí kombinačních obvodů
        - Sestavíme logickou funkci odpovídající řízení, kterou reprezentujeme kombinačními obvody
- Exception
    - Vnitřní příčina
    - Arithmetic overflow
    - Invalid instruction
- Interrupt
    - Vnější příčina
    - IO
- Vícecyklová datová cesta
    - Realizace řadiče
        - Paměť + kombinační obvod
- Slidy 64-73 jsou fakt hovna

06-zvysovani_vykonnosti.pdf

- V délce taktu pipeline se musí stihnout nejdelší fáze pro nejsložitější instrukci
- Latence instrukce = délka cyklu
- Slide 5 nějaké vzorečky
- Multiple issue pipeline
    - Potřebujeme pomocné sčítačky, ALU nelze paralelně sdílet
    - Obvykle se používá řízení z jednocyklového řadiče
- Flynn's bottleneck
    - Teoretické omezení skalární pipeline CPI = 1
    - Avšak vždy bude platit, že CPI > 1, protože stalls a režie než naplníme pipeline
- Zvyšování počtu fází pipeline od určité délky způsobuje **pokles** výkonu
- Dnešní procesory mají obvykle 4 pipeliny, každá má 14-19 fází

- Out-of-order execution
    - Zachováváme data flow programu
        - Tváříme se, že se program vykonává sekvenčně
    - Instrukce, která se dostane dříve do pipeline než nějaká jiná ji může opustit později
    - Obvykle použití s dynamickým plánováním a spekulativním prováděním instrukcí pro nejvyšší využití ILP
    - Kolize mezi jmény registrů v instrukcích
        - WAW (Write After Write)
        - WAR (Write After Read)
        - RAW (Read After Write)
        - První dvě jdou vyřešit přejmenováním registrů
- Historické priority
    - Rozsáhlé instrukční sady, složité operace
    - Překlenutí sémantické mezery mezi assemblerem a vyšším programovacím jazykem
- Aktuální priority
    - Malé instrukční sady, jednoduché instrukce
    - Rychlejší provádění instrukcí, snadnější optimalizace
- Proč je MIPS assembler jednoduchý (na pipelinening)
    - Instrukce stejné délky
    - Malý počet formátů instrukcí
    - Přístup pouze na zarovnané adresy

07-pametova_hierarchie.pdf

- **Paměťová zeď**: Výkon procesorů omezen výkonem pamětí
    - Jednoduché operace (bez přístupu do paměti) trvají **desetiny** ns
    - Operace s přístupem do paměti trvají **desítky** ns
    - **Nedosažitelný cíl**: paměť stejně rychlé jako procesoru
- Kapacita paměti × Rychlost paměti
    - Vzájemně se vylučuje
- Jak se vypořádat s paměťovou zdí
    - Princip lokality přístupu do paměti
    - Vlastnost většiny reálných programů
    - Časová lokalita
        - Věci, které jsme použili se budeme spíš vracet
    - Prostorová lokalita
        - Věci, které se v paměti nacházejí kousek od věci, kterou jsme použili dost možná použijeme
    - K datům, které pravděpodobně využijeme uložíme do menší rychlejší paměti
- Z obyčejného klopného obvodu typu D jsme schopni vyrobit rozumnou SRAM

    - V přímočaré implementaci by 1 bit $\approx$ 9 tranzistorů
- Dvojící invertorů + řídícími tranzistory dosáhneme na 1 bit $\approx$ 6 tranzistorů
- SRAM v maticovém uspořádání ($M \times N$)
    - Výběr řádku — dekodér 1 z M
    - Čteme sloupce
- DRAM
    - Kondenzátor + řídící tranzistor
    - Informace je uložena v kondenzátoru v podobě nabití
    - Kondenzátor se však samovolně vybíjí
    - Čtení je destruktivní → čtená hodnota se hned zapíše zpátky
- Hiearchie paměťových komponent
    - Vyšší vrstvy: rychlé, malé, drahé
    - Nižší vrstvy: pomalé, velké levné
    - Vzájemné propojení sběrnicemi
- Cache
    - Iluze velké a rychlé paměti
    - Přesun dat mezi vrstvami řídí hw
        - Cache controller (radič cache)
    - SRAM
    - Software — kompilátor může dávat nápovědy
- Obsluha výpadku cache
    - Cache controller si vyžádá data z následující úrovně hiearchie na základně adresy výpadku
    - Zapíše data, tag, (valid bit) do řádku cache
- Výpadky cache zpožďují pipeline
- Výkonnostní metrika
    - $t_{avg} = t_{hit} + t_{miss} \cdot \%_{miss}$
- Snížení miss rate
    - Zvýšit celkovou kapacitu cache
    - Reorganizovat cache
        - Velikosti bloku $\times$ Velikost celé cache
- Zvětšení velikost řádky
    - Využití prostorové lokality
    - Snížení miss rate **!do určité míry!**
    - Potenciálně zbytečné přenosy
    - Předčasná náhrada užitečných dat
    - Cache fill time
        - V principu by měl trvat déle s větším řádkem
        - Avšak používáme *early restart* nebo *critical word first*
        - Early restart
            - Execution pokračuje s načtením slova, které jsem požadoval (zbytek cache liny se ještě kopíruje)

        - Critical word first
            - Vylepšení early restart o to, že jako první načtu slovo, které právě potřebuji, aby execution mohl co nejdříve pokračovat (zase zbytek cache se potom ještě kopíruje)
        - Fill time se z tohoto důvodu v praxi příliš nemění s velikostí bloku
- Asociativní mapování cache
    - Prodlužuje hit time (musím hledat ve více řádcích)
- Důsledky 3C modelu
    - Asociativita
        - Snižuje počet konfliktních výpadků
            - Hlavně při přechodu od 1 way do 2 way, vyšší stupně associativity nemají přílišný význam pro větší cache
        - Prodlužuje hit time (více řádků, kde data mohou být)
            - Zas ne tak moc, obvykle používáme velké množství paralelních komparátorů, které se snaží toto eliminovat
    - Velikost bloku (cache line)
        - Vesměs neovlivňuje hit time
            - Early restart, critical word first
    - Kapacita
        - Prodlužuje hit time
        - Snižuje počet kapacitních výpadků
- Write miss v případě write-through cache
    - Je možné zároveň číst tag a zapisovat data
    - Pokud přepíšeme špatná data (tag nesedí), tak správná data jsou ještě v paměti
    - Některé procesory umožňují nastavit strategii zápisu (write allocate nebo write no allocate) na úrovni jednotlivých stránek
- Write miss v případě write-back cache
    - Není možné zároveň číst tag a zapisovat data
    - Pokud by tag neseděl a řádka by byla dirty, **přišli bychom o data**
    - 1. možnost zápisu
        - Kontrola hit/miss, potom zápis
    - 2. možnost zápisu
        - Kontrola hit/miss s uložení dat do **store bufferu**
        - Potom se data ze store bufferu zapíší
    - Pokud máme vyhodit z cache řádku s dirty bitem tak se zapisuje:
        - Nejdřív do **write-back bufferu**, z něhož se data zapíšou do paměti/nižší vrstvy
    - **Při hledání v cachi nutno prohledávat i write-back buffer**
- Multilayer cache
    - Chceme snížit penalizaci při výpadku
    - Různé úrovně cache mají různé role
    - Primární cache
        - Miminalizace hit time

        - Menší kapacita, menší velikost řádku
    - Sekundární cache
        - Snížení miss rate
        - Výrazně vyšší kapacita, vyšší asociativita
- Shrnutí
    - Paměťová zeď
    - Neexistuje ideální paměťová technologie
    - Lokalita přístupu do paměti
    - Hiearchie pamětí
- **Cache jako iluze ideální paměti**

- Víceprocesorové systémy přinášejí **mnoho problému**
- Očekávané chování
    - Čtení z nějaké adresy v paměti vždy vrátí hodnotu, která byla na tuto adresu naposledy zapsána libovolným procesorem
- Moderní procesory replikují obsah paměti v lokální cache
- V důsledku zápisů mohou mít procesory **různé** hodnoty pro **stejnou** adresu v paměti
- Koherentní paměťový systém
    - **Procesor vidí vlastní zápisy v programovém pořadí**
    - **Zápisy do paměti nakonec uvidí všechny procesory**
    - **Zápisy do stejného místa jsou uspořádané**
        - Všechny procesory uvidí zápisy do stejného místa ve stejném pořadí
- Koherence
    - Určuje **jaké hodnoty** uvidíme při čtení
- Konzistence
    - Určuje **kdy** budou zápisy viditelné pro čtení
- Zajištění koherence
    - HW zajistí, že čtení nějakého místa v paměti libovolným procesorem vrátí “poslední” hodnotu zapsanou do tohoto místa
    - Řešení založena na zneplatnění nebo aktualizaci dat v cache (**Koherenční protokol**)
- Snadné řešení: sdílená cache
    - Problémy s rušením procesorů
- Snooping
    - Všechny události související s koherencí jsou šířeny (broadcastem) všem procesorům (cache controllerum) v systému
    - Cache controllery individuálně reagují, tak aby byla zajištěna koherence dat
- Jednoduchý koherenční protokol — Valid/Invalid (VI) protokol
    - Při zápisu invalidujeme odpovídající cache line ostatních procesorů
    - Použití s write-through
    - Ostatní procesory při čtení cache miss a načtou data z paměti
- Pro efektivnější použití s write-back potřebujeme sofistikovanější koherenční protokol (**MSI**)

    - Cache line je v **exclusive** stavu, pokud je pouze v jedné cache line
    - Pouze tuto cache line je možné modifikovat
    - Pokud procesor chce modifikovat cache line, která není v **exclusive** mode, tak nejprve pošle požadavek na **exclusive read**, ostatní procesory musí zareagovat tím, že si danou cache line invalidují
- MESI
    - Vylepšené MSI
        - Zde jsou potřeba 2 transakce pro modifikaci dat
    - Řešení: nový stav E (exclusive clean)
- Existují mnohé lepší a efektivnější protokoly
    - MOESI (AMD Opteron)
    - MESIF (Intel)
- Důsledky implementace koherence
    - **Každá cache musí poslouchat a reagovat na koherenční události šířené po sdíleném spoji**
    - **Vyšší zatížení sdíleného spoje**
    - GPU koherenci neimplementují vůbec nebo jenom v omezené formě

- ***Co je špatně na následujícím kódu?***

```
// allocate per-thread accumulators
int counters [NUM_THREADS];
```

- ***Lepší verze***

```
// allocate per-thread accumulators
struct PerThreadState {
    int counter;
    char padding [64 - sizeof (int)];
}

PerThreadState counters [NUM_THREADS];
```

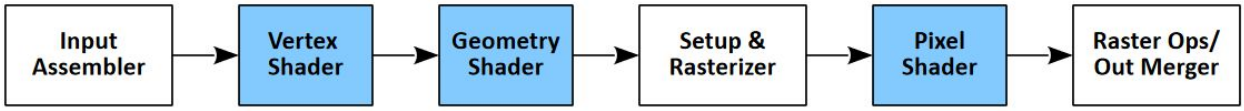
- False sharing
    - Dvě vlákna zapisují do různých proměnných ve stejně cache line
    - Cache line přeskakuje mezi cache zapisujících procesorů
    - Koherenční protokol způsobuje velké množství komunikace, přestože spolu vlákna vůbec nekomunikují

    - Veškerá komunikace je však pouze nežádoucí produkt **falešného sdílení**
    - Může výrazně ovlivnit výkonnost programu na architekturách implementujících koherenci (všechny CPU)

08-paralelni_zpracovani.pdf

- Dosavadní metody paralelizace
    - Na úrovni instrukčních kroků
        - Pipelining
    - Na úrovni instrukcí
        - Multiple issue
- Další metody paralelizace
    - Na úrovni procesů
    - Na úrovni dat
- Proč je těžké psát paralelní programy?
    - Musí být rychlejší než sekvenční
    - Škálování s počtem procesorů
    - Ambdahlův zákon
- Správné rozkládání zátěže (load balancing)
- Flynnova taxonomie
    - Nejznámější rozdělení paralelních výpočetních strojů
        - Podle počtu *instrukčních proudů*
        - Podle počtu *datových proudů*
    - SISD
        - Single instruction stream, single data stream
        - Normální jednojádrový procesor
    - SIMD
        - Single instruction stream, more data stream
        - Vector processors
            - Místo skalárů pracuje s vektory
            - Např. add by sčítalo vektory
        - Dnešní procesory mají takové instrukce
            - SSE
            - MMX
            - 3DNow!
    - MISD
        - Multiple instructions, single data stream
        - Více instrukcí pracuje na jedněch datech
        - Stream processors?
        - Příliš se nepoužívají takové procesory

    - MIMD
        - Normální dnešní procesor, který obsahuje několik procesorů, jenž pracují na vlastních datech
- Hyper-threading u Intelu
    - 2 jádra sdílí L3 cache
- Dneska specializované koprocesory pro nějaké specifické úkoly
- Na některé specifické instrukce jsou dneska mnohem rychlejší grafické karty
    - Zadrátované grafické algoritmy do vhodných instrukcí
- Popř. FPGA
- Programovatelný zásah do grafických pipelines



- Odlišnosti v cílech návrhu CPU a GPU
    - CPU je více univerzální (musí zvládat různé typy úloh)
- Shrnutí
    - *Správné fungování vyšších vrstev abstrakce závisí na správném využití nižších vrstev abstrakce*
    - *Základy vnitřního fungování procesoru se dají snadno pochopit*
    - *Občas je dobré podívat se do historie*
    - *Moorův „zákon“ není zákon*
    - *Zachytit výkonnost počítače nelze jedním číslem*
    - *Počítače jsou složeny jako stavebnice*
    - *Aplikační binární rozhraní (ABI)*
    - *Optimalizace má největší účinek pro nejčastější případy*
    - *Když nelze zvýšit hrubý výkon, je možné zvyšovat propustnost*