Jiný pohled na cache

Algoritmy a cache

- Přístupy do paměti v algoritmech jsou dvou druhů
  - S předvídatelnou adresou
    - Lineární průchody polem

```
for ( i = 0; i < N; ++ i ) { /*...*/ a[ i] /*...*/ }
```

    - Lineární průchody s větším skokem

```
for ( j = 0; j < M; ++ j ) for ( i = 0; i < N; ++ i ) { /*...*/ a[ i][ j] /*...*/ }
```

  - S "náhodnou" adresou
    - Hashovací tabulky

```
for ( i = 0; i < N; ++ i ) { /*...*/ a[ hash( d[ i])] /*...*/ }
```

    - Bucket-sort

```
for ( i = 0; i < N; ++ i ) { /*...*/ a[ b[ i]] /*...*/ }
```

    - Binární vyhledávání

```
while ( /*...*/ ) { if ( a[ j] > /*...*/ ) j = /*...*/; else j = /*...*/; }
```

    - Spojové struktury

```
while ( p != 0 ) { /*...*/ p = p->next; /*...*/ }
```

- Přístupy s předvídatelnou adresou
  - Efekt řádku cache: Husté lineární průchody mají dobré hit ratio
  - Write buffers: Zápisy obvykle nezdržují
  - Hardware prefetching
    - procesor detekuje lineární průchody a načítá data do L1 předem
  - Software prefetching
    - překladač generuje instrukce pro přístup k datům předem
      - běžné instrukce pro čtení - vyžadují jistotu příští iterace
      - speciální instrukce pro spekulativní čtení - potlačené výjimky
    - totéž může udělat programátor ručně
      - u dnešních procesorů/překladačů nebývá zapotřebí
  - Latence přístupu se skryje paralelním vykonáváním jiné užitečné činnosti
  - Rozhoduje prostupnost sběrnic paměť-cache-ALU (bandwidth)
    - Algoritmy se optimalizují na nejlepší využití dané prostupnosti

Algoritmy a cache

- Přístupy s "náhodnou" adresou
  - Adresa nezávislá na předchozí iteraci
    - Latenci přístupu lze skrýt paralelizací
      - Někdy to dokáže sám překladač
    - Hashovací tabulky

```
for ( i = 0; i < N-1; ++ i ) { x = hash( d[ i+1]); /*...*/ v /*...*/; v = a[ x]; }
```

    - Bucket-sort

```
for ( i = 0; i < N; i += 2 ) { /*...*/ a[ b[ i]] /*...*/ a[ b[ i+1]] /*...*/ }
```

  - Adresa závislá na předchozí iteraci (loop-carried dependence)
    - Paralelizovat není s čím
    - Rozhoduje latence přístupu
    - Binární vyhledávání

```
while ( /*...*/ ) { if ( a[ j] > /*...*/ ) j = /*...*/; else j = /*...*/; }
```

    - Spojové struktury

```
while ( p != 0 ) { /*...*/ p = p->next; /*...*/ }
```

Algoritmy a cache

- Adresa závislá na předchozí iteraci (loop-carried dependence)
  - Paralelizovat není s čím
  - Vyžaduje globální úpravu algoritmu (změny rozhraní funkcí)
    - Výměna vzájemné vnořenosti cyklů
      - loop reversal; obecněji afinní transformace cyklů (loop skewing)
    - Vyžaduje stabilní počet iterací vnitřního cyklu
    - Binární vyhledávání

```
for ( i = 0; i < N; ++ i ) bsearch( a, M, b[ i]);
```

      - upraveno na

```
bsearch_many( a, M, b, N);
```

  - U nevhodných datových struktur paralelizovat nelze
    - Překážkou je nevyváženost počtu iterací
  - Paralelizace zhoršuje lokalitu přístupů do paměti
    - Skrytí latence za cenu sníženého cache hit ratio

Ideální algoritmus

- Celková architektura „ideálního algoritmu“
  - Jádro úlohy pracující v registrech (podúloha velikosti 32-512 B)
    - Pouze lokální proměnné, pokud možno žádné pole
    - Proměnné čteny z paměti na začátku/zapisovány do paměti na konci
    - V ideálním případě SIMD instrukce
  - Podúlohy do velikosti 8-16KB
    - Data se vejdou do L1
    - Data podúlohy mohou být v paměti mírně nesouvislá
      - Každý blok násobkem 64 B (cache line)
      - Jsou-li bloky vzdálenější než 4 KB, pak nejvýše 32 bloků (TLB1)
    - Podúloha řešena iterativně nad jádrem úlohy
      - Rekurzivní řešení mívá příliš velký overhead
      - Iterace umožňuje prefetch
  - Úlohy větší než 16 KB
    - Řešeny rekurzivně metodami Cache-Oblivious algoritmů
      - Obvykle se dělí na dvě podúlohy o poloviční počtu operací
      - Každá podúloha má ***větší*** než poloviční spotřebu paměti
      - Vybírá se takový způsob dělení, který minimalizuje paměťový překryv podúloh
      - Okolo 16 KB se rekurze nahradí iterací podúlohy
    - Data každé podúlohy by měla mít malý počet bloků (problém TLB)

Algoritmy a cache - další pohled

- Přístup na náhodné adresy
  - Schopnost přístupu na náhodné adresy je pro algoritmus klíčová
    - bsearch, hash,...
  - Nalezení příslušné buňky paměti je součástí užitečného výkonu algoritmu
    - Program vykonává užitečnou práci pomocí adresních dekodérů paměti
      - Adresní dekodéry jsou v paměti pořád - zaměstnejme je!
      - Paměť má nezávisle pracující bloky - zaměstnejme je paralelně
- Přístup na předvídatelné adresy
  - Předvídatelný (lineární) přístup nevyužívá schopnosti RAM
    - Adresní dekodéry opakovaně dekódují podobné adresy
      - Zbytečný hardware, zbytečná spotřeba energie
  - Architektura RAM stroje je pro takové algoritmy nadbytečná
    - Běžné programovací jazyky jsou této architektuře podřízeny
  - Vystačili bychom s Turingovskou páskou
    - Neumíme ji fyzicky realizovat
    - Neumíme v tomto prostředí programovat