

Algoritmy a jejich implementace

Vyučující: Mgr. Martin Mareš Ph.D., Cvičení: Filip Štědranský

15. dubna 2021

Obsah

1	Uvod, Cko	2
1.1	Uvod	2
1.2	Cko	2
1.3	Preprocesor	6
1.4	Bitové triky	7
2	Hardware architecture	9
2.1	Historický vývoj architektur	9
2.2	x86	9
2.3	AMD64	12
2.4	Disassembler	12
3	Hierarchie pameti	12
3.1	Memory Management Unit	13
3.2	Cviko	13
4	Profilování, peft, cachegrind	13
4.1	Zajímavé perf udalosti	13

1 Uvod, Cko

1.1 Uvod

- V tomto předmětu se zaměříme na PC hw architekturu.
- Cílovým OS je Linux, jelikož běží na většině super počítačů, klastrů apod. Ostatní OS stavěné na výkon se chovají podobně.
- Cko jako cílový jazyk (bohužel občas místo kompilátoru je komplikátor). GCC jako cílový kompilátor.

Příklad 1.1. Motivační příklad: Transpozice matic. //TODO obrázek

a) Křivka naivního algoritmu roste (proč? na RAMu by neměla) + spiky.

- Na začátku skoky lze vysvětlit velikostí $L3$ cache, ale ne všude. Hlavně spacial lokality v hlavní paměti.
- skoku u 2^i protože cache není plně asociativní. Z množiny čísel se stejnými spodními bity, jenom málo se může dostat to cache na stejné místo. Podobně u dělitelů mocnin 2ky.

b) Je to naivní algoritmus + snažíme se vyhnout mocninám 2ky tak, že přidáváme konstantní mezery mezi řádky. Znázorňuje zelená křivka. Proč roste jako exp na nejmenších velikostech? Vysvětlíme později.

c) Tiling algoritmus

d) Transpose and swap (cache oblivious algoritmus). Chová se o něco líp než tiling.

Poznámka 1.2. Dotaz: má smysl při inicializaci algoritmu zjistit velikost cache?

Obecná odpověď je spíše NE, ze 2 důvodů:

1. Reálná hierarchie může být extra složitá. Jenom velikost nepomůže.
2. U Transpose and Swap větší role hraje režie rekurze. Protože rekurze stojí víc než přístup do paměti. Takže se musíme zastavit na takové největší velikosti podproblému, který se vejde do cache. Pokud víme, že $L_1 > 16kb$, tak funguje matice 64×64 .
3. Nastavení parametrů může škodit kompilaci. Jde ale obejít tak, že zkompilujeme několik verzí algoritmu s různými parametry, pak rychle zjistit který je rychlejší a nadále používat.
Např Linux kernel má k dispozici 4 algoritmy pro sw REID. Při bootu zkontroluje který je rychlejší a bude používat po celou dobu.

1.2 Cko

Tady budou pouze komentáře k slajdům Uvod do Cka.

Vývoj dialektů Cka

1. K&R Cko, původní návrh popsáný v knize.
2. První standardizace, ANSI a ISO. Je známá jako C89. Z ní se vytvořili dialekty dle překladače a taky C++.

3. C99 nejlepší featury z dialektů. Pak vznikla ještě C11 verze a C17 (druhá jen opravovala chyby první).
4. Občas Cko přebralo featury zpět od C++, např v C11 memory management pro paralelní programování.

Číselné typy

1. Není specifikováno, jestli **char** je *signed* nebo *unsigned*.
2. Typ **bool** máme protože kompilátor lepe optimalizuje. Taký lepší semantický význam. Jmenuje se ale **__Bool** aby kompilace nerozbila stávající programy, které již měli bool přes *typedef*. Pokud ale chci právě **bool**, stačí použít *include <stdbool.h>*.
3. stejně jako v C++, existují typy pevné délky, např u `__int32`. Hlavně se používají u síťových protokolů, taky inline assembler.

Union má pouze jednu ze dvou položek v paměti, sám neví který to je. Je potřeba odvodit z kontextu.

Částečné typy

1. `struct who_knows_what *`
2. `int f()` - nevíme parametry
3. `int f(void)` - nemá žádný parametr
4. `int []` - pole neznámé velikosti

Reprezentace v paměti

1. pozor na bitová pole při paralelním programování. (necelé byty se hlavně používají hlavně pro síťové pakety)
2. pozor 2kový doplněk není zaručený.
3. C11 má operátor `__AlignOf(type)`, nejčastěji velikost typu.
4. překladač nesmí prohodit pořadí položek uvnitř. Protože se zaručuje, že 2 struktury zděděné ze stejného typu mají stejný prefix v paměti.

Literály

1. Pozor, přidání nuly na začátek změní literál na 8 soustavu. Změní se i hodnota.
2. Veškeré celočíselné počítání se provádí v typu `int`.
3. Floating-point literály bez ztráty přesnosti se zapisují s exponentem, například `0x1.ffep10`, kde `p` je oddělovač pro exponent.

Operátory

- používáme `pointer->item`, protože při zápisu `*pointer.item` tečka má větší prioritu, museli bychom psát `(*pointer).item`
- `a = x ++` pak `a` má původní hodnotu. `a = ++ x` pak `a` má hodnotu po inkrementu. `x = x ++` je hloupost, není definované.
- Hádanka `a+++++a = ??`
- nepoužívat bitové operace u znaménkových typů
- `!!x` je přetypování na `bool`
- nemodulit záporným číslem
- `float` není lineárně uspořádané. `NaN != NaN`.
- počítání s `unsigned` a `signed` dává `unsigned`
- nepsat `printf("%d%d%d", a++, a++, a++)`. Dobrým zvykem je měnit hodnotu proměnné pouze jednou v příkazu.

Nedefinované chování - cokoliv se může stát.

Pointry

- `pointer` na pole je něco jiného než `pointer` na první prvek. S `pointrem` na pole se ale běžně nepracuje.
- `pointer` na fce se zapisuje `int(*f)(int a);`
- nepoužívat složitější konstrukce jako `pointer` na `pointer` poli `pointrů`.

Modifikátory

- u `const` záleží kde je. Protože

```
const int *x
```

je konstantní integer, když

```
int * const x
```

je `const` `pointer`.

- `register` `int x` - je historický, hint překladači, že může uložit do registru místo paměti. Pak ale nejde použít `pointer`. Dneska se nepoužívá, překladač optimalizuje lepe.
- `volatile` (hist že se může nečekaně změnit). Třeba namapovaná periferie do paměti. Třeba nějaký port cpe data, pokud nenapišeme `volatile`, tak překladač může vyoptimalizovat 2 čtení na 1. Lze taky použít pro zrušení optimalizace pro měření.
- `restrict` - na adresu lze přistupovat pouze tímto `pointrem` nebo odvozeným. Pak překladač lepe optimalizuje. Např lze použít při kopírování paměti, zaručí že 2 useky se nepřekrývají. Používá to funkce `memcpy` nebo `memmove`.

- `static int x` - u globální jako `private`. Je viditelná uvnitř modulu.
- `extern int x` - společná proměnná pro několik modulu, stejný název.
- `inline` - místo volání se vloží. Dneska překladač um sám.

Příkazy

- prázdný příkaz se může hodit. Ale je potřeba dávat pozor

```
while(...);
```

zopakuje prázdný příkaz do nekonečna.

Tady je ale potřeba napsat středníky, nebo `{}`

```
if(...) a;
else b;
```

- výraz je příkaz (vyhodnotí se side efekty)
- `for(a, b, c)` cyklus je ekvivalentní tomuto

```
a;
while(b)
{
    příkazy;
    c;
}
```

- příkaz `goto` je potřeba použít v následujících případech
 - `break` ze 2 a více cyklů.
 - konečné automaty
- jde zkombinovat `while` a `switch`

```
switch()
{
    while(b)
    {
        case 1:
            ;
    }
}
```

Hodí se třeba pro zpracování poli po 4. `switch` je pro `mod 4`

- `__static_assert` kontrola v čase kompilace, že např integer je nějaké velikosti

1.3 Preprocesor

- proměnlivý `#` argumentů, makro `< stdarg.h >`. Taky makro `va_arg`.
- trigrafy, používat nechceme, historická záležitost, hlavně nepsat omylem `??` char.
- `$include` preprocesor nahrazuje kódem jiného souboru. Preprocesor pracuje na pp-tokenech
- gcc umí tzv linked optimisation - optimalizace mezi moduly.

Makra Jsou 3 druhy

1. s závorky (má argumenty). Např

```
# define P1(x) x + 1
```

2. s závorky (bez argumentů)

```
# define Z() null.
```

3. bez závorky (závorky nebudou částí expanze)

```
# define X x.
```

Vlastnosti 1.3.

Makra

- je zakázáno definovat stejné makro 2x. Undef ale jde provést libovolně krát. Proto v souborech často guardy.
- dle konvence `<header>` hledá mezi systémové soubory. *"header"* - soubory definované programátorem.
- `#` se používá při generování kodu třeba pro vypsání chybové hlášky aby odkazovalo na původní soubor.
- `#pragma` jsou specifické pro kompilátor.
- při aritmetice v makrech je potřeba používat závorky.

Ostatní Nezařazené komentáře

- `size_t` určuje velikost objektů.
- od C11 máme vnořené (lokální) funkce. Pro implementace se používá tzv trampolína (místo v paměti kde se ukládají data)
- kod pro defaultní hodnotu

```
a?:b = a ? a : b
```

- range v case


```
case 1...5:
;
```

- pokud jeden case propadá v další je konvenci psat komentář, jinak warning od kompilátoru

```
case 1:
    ...
    // fall through
case 2:
```

- `__auto = 5`, stejně jako `var` v C#.

GCC atributy

- `hot` (hodně optimalizovat, hot path)
- `cold` (málo optimalizovat, moc se nespouští)
- `noinline` (hlavně pro profilování)
- `flatten` - inline všechno co se volá uvnitř. Na co se hodí??
- `packed` - nepřidávat padding

1.4 Bitové triky

Bitové triky mimo jiné umožňují vyhnout se podmíněným skokům, tzv branchless programování. Což hodně pomáhá branch predictor.

Cvičení 1.4. 1. bitová rotace

2. násobení konstantou
3. zaokrouhlení na nejbližší násobek 16
4. test zda je číslo mocnina 2ky
5. zaokrouhlit na nejbližší mocninu 2ky.
6. zrcadlení: bitová i bytová
7. abs hodnota
8. signum
9. parita, lichý nebo sudý # jedniček

Důkaz. 1.

$(x \gg k) \mid (x \ll (32 - k))$

2. Rozepsat jako součet mocnin 2ky, pak

$(x \ll 4) + (x \ll 2) \dots$

3. Potřebujeme vynulovat spodní 4 bity

```
x & 1 ... 110000
```

Postup jak vyrobit masku

```
~(1 << 4) - 1
```

Pro zaokrouhlení nahoru

```
(x + 15)/16
```

uprostřed

```
(x + 8)/16
```

4. Potřebujeme najít první jedničku vpravo

```
x & ~(x - 1) == x
```

Pokud máme zaručený 2kový doplněk, tak

```
x & (x - 1) == 0
```

5. Máme v registru 0000x, kde x je posloupnost bitů začínající 1. Vyrobíme stejně dlouhou posloupnost jedniček.

```
x -= 1; //abychom dostali neostrou nerovnost
```

```
x |= x >> 16;
```

```
x |= x >> 8;
```

```
...
```

```
x |= x >> 1;
```

```
x += 1; //čímž dostaneme chtěnou mocninu 2ky
```

Pro dolní odhad na začátku přidáme

```
x = (x >> 1) + 1;
```

6. Divide and Conquer

```
(x >> 16) | (x << 16)
```

```
((x & 0xff00ff00) >> 8) | ((x & 0x00ff00ff) << 8)
```

```
...
```

7.

```
y = x >> 31; (aritmetický posun)
```

```
(x ^ y) - y
```

Pro kladná y je same nuly. Dostaneme x. Pro záporná

$$x^y = \sim x = (-x - 1)$$

$$(-x - 1) - y = (-x - 1) - 1 = -x$$

8. První řešení

```
(x > 0) - (x < 0)
```

Druhé

```
((unsigned)x) >> 31
```

9. Divide and Conquer

```

x ^= (x >> 32);
x ^= (x >> 16);
x ^= (x >> 8);
x ^= (x >> 4);
x ^= (x >> 1);
// máme XOR všeho

```

```
x &= 1;
```

10. Swap přes XOR. Funguje protože XOR je involuce

```

a ^= b;
b ^= a;
a ^= b;

```

11. Jednostranný spojак přes XOR. Místo pointeru na next a prev uložíme

$prev^{next}$

Posouváme se pomocí XORu. □

2 Hardware architecture

2.1 Historický vývoj architektur

Historicky procesor 8086 → x86 → AMD64.

U x86 nás zajímá 32b mod s lineární adresací. DOS by fungoval na stejném procesoru trochu jinak.

2.2 x86

Datové typy

- integer 8/16/32 a nějaké operace s 64b integery.
- float
 - a) 32 (24 + 8) - float
 - b) 64 (53 + 11) - double
 - c) 80 (64 + 16) - long double
 - d) jen občas 16 (11 + 5) pouze ve vektorech. Používá se v grafice
- vektory 128b (homogenní, všechno kromě long double jako základní typ)

Paměť Z pohledu programu pole bytů. Von Neumann architektura, společná pro kod a data. Ne všechna částí paměti lze používat. Když šáhneme na zakázanou část, tak OS zabije.

Přístup na špatně zarovnaný vektor selže!

Registry Univerzální registry, lze dělat skoro cokoliv.

- EAX
Původně na 8086 byl registr AX, spodní část AL, horní AH.
Pak přidali ještě 32 bitů.
- EBX
- ECX
- EDX

Registry co složili pro cílovou adresu. Dnes jsou univerzální, ale na rozdíl od EAX nelze použít dolní částí.

- ESI - source index
- EDI - destination index
- EBP
BP - base pointer. Ukazuje dle konvence na oblast v paměti kde jsou lokální proměnné a parametry. Dnes už je univerzálním.
- ESP
SP - stack pointer. ESP ukazuje na poslední adresu v Stacku.
- EIP - instruction pointer.
- EFLAGS - každé bity mají jiný význam.
Např zero - poslední aritmetická operace byla 0.
Přetečení (znaménkové a bezznaménkové)
Řidiší bity, např D - direction.

Floutové registry

- ST0 - ST7, tvoří float stack. Dnes se moc nepoužívá
- FPUSR (FPU status registr) - jako flagy.
- FPUCR (FPU control registr) - řídí jestli při dělení 0 mu vyskytnout výjimka apod.

Registry vektorové jednotky

- XMM0 - XMM7 128 bitové
- YMM0 - YMM7 256 bitové
- MXCSR (control and status) - další flagy.

Poznámka 2.1. Lze oddělovat Int, Float, Vector část procesoru.

Assembler Intel (windows) vs AT&T (linux).

Např přičtení 1 k registru EBX

- Intel: ADD EBX, 1
- AT&T addl \$1, %ebx
kde l znamená (long), q by bylo 64 atd.
před literálem je dolar, jinak by jednička znamenala místo v paměti.

Např přičtení 1 k místu v paměti uložené registru EBX

- Intel: ADD DWORD PTR [EBX], 1
- AT&T addl \$1, (%ebx)

Úmluva 2.2. Jelikož jsme na Linuxu, budeme používat AT&T.

Operandy instrukce

- \$ číslo
- %registr
- číslo - adresa v paměti
- (%reg) - adresace registrem
- číslo (%registr) - $adresa = reg + číslo$
- číslo (%r1, %r2, mul) - $adresa = r1 + mult * r2 + číslo$
mul je 1,2,4,8.
hlavně se hodí u adresaci poli.

Nechceme psát v assembleru, ale rozumět co vygeneroval překladač. Pokud je neefektivní, chceme mu pomoci kod zlepšit.

ABI ABI - application binary interface, specifické pro architekturu, OS a jazyk (C). Ostatní jazyky umí linkovat C-kovou knihovnu.

Říká např kde je program v paměti. Ale nás toto nezajímá.

Např volací konvence. Bylo by dobrý předávat parametry v registrech. Toto obecně nejde, protože registrů je málo a můžou se použít pro jiné věci. Proto je dohodnuté že volající přidává parametry na stack. Dolu je první argument. Je potřeba pro funkce s proměnlivým počtu argumentů.

Někde na zásobníku je návratová adresa. Nahoře od ní je na starosti Volajícího. Pod ní Volaný.

Definice 2.3. Stack frame -

Výsledek funkce se vrátí buď v EAX nebo EAX/ADX (pokud větší než 64). Float v ST0. Pokud struktura, tak výjimka, volaný uloží v paměti. Posílá adresu v paměti.

Scratch Věci co může měnit volaný:

- EAX, ECX, EDX
- ST0-ST7
- část EFLAGS.

Zbytek musí zůstat jako dříve.

EBX není scratch, protože se používá pro sdílené knihovny, musí pracovat s relativní pamětí (nahrávají se kamkoliv). EBX ukazuje kde jsou data sdílené knihovny.

2.3 AMD64

Změny oproti x86:

- EAX se rozšíří na horních 32 bitů → RAX.
Analogicky pro EBX, ECX, EDX, ESI, EDI, EBP, ESP.
Jsou univerzální proto se jmenují R0-R7.
Pozor, 16 bitové instrukce jsou pomalé.
- Nově máme R8-R15.
- FPU se moc nezměnila, jen veškeré počítání jsou vektorové. Pozor nemám 80 bitový float, na toto se stále používá x86.
- Nově XMM8-15, YMM8-15.

Jelikož registrů je hodně, parametry se přidávají v registrech. Pokud se nevešlo - tak na zásobníku. Celkem 6 celočíselných argumentů v registrech, 8 float argumentů.

2.4 Disassembler

3 Hierarchie pameti

Definice 3.1. Dynamická paměť - jeden kondensátor.

Každé čtení je destruktivní, takže čtení musí zároveň zapsat. Musíme taky obnovovat data dejme tomu každé 100ms.

Paměť je taky omezena kvůli rychlosti světla, paměť je moc daleko od CPU. Pokud máme 1Gz procesor, tak během jednoho taktu světlo uletí 30cm.

Typy cache

1. Plně asociativní cache. Zavedeme cache line (bloky). V praxe se nepoužívá, protože potřebujeme hodně HW pro porovnávání a shromažďování dat.
2. Přímo mapovaná cache
Nevýhody: (cache aliasing) viz úvodní příklad násobení matic při velikosti řádku 2^i .
3. Kvůli nevýhodám 1, 2 používá se kompromis - Množinově asociativní cache. Plně asociativní uvnitř podmnožin.
4. Dalším vylepšením je hierarchie cachi. Aby cache byla rychlá, musí být blízko procesoru, takže musí být malá. Proto první cache je malá, druhá je větší.

Dotaz: menší instrukční sada, znamená lepší cachování v *L1I*.

3.1 Memory Management Unit

Poznámka 3.2. Pozor –O3 optimalizaci není rozumné zapínat globálně, ale jenom lokálně kde očekáváme, že může výrazně zlepšit.

3.2 Cviky

Příklad 3.3. Při překladu prohození pole překladač nepočítá 2. index protože si pohopil, že procházíme pole sekvencně. Takže si pořídil 2 pointery které synchronně incrementuje.

Příklad 3.4. V příkladu sqdiff najdeme bitový trik pro absolutní hodnotu.

Příklad 3.5. Najdeme zneužití instrukce **adcl** pro podmíněné přičtení jedničky.

Příklad 3.6. Vypočet Fibonacci. Všimneme si, že -O3 nešáhá do paměti, všechno počítá v registrech.

Pak místo +4 pro další prvek v pole, si pořídíme další pointer který ukazuje na 2. prvek v poli.

Příklad 3.7. V příkladu vypočtu polynomu při –O3 nejsou žádné cykly. Proč? Protože kompilátor si všiml, že pole má pevnou velikost, proto cyklus byl úplně rozbalen.

4 Profilování, peft, cachegrind

4.1 Zajímavé perf udalosti

- TLB
 - dTLB-load-misses / dTLB-loads
 - iTLB-load-misses / iTLB-loads
- Počty cache misses
 - L1-dcache-load-misses
 - LLC-load-misses # last-level cache, i.e. L3
 - LLC-store-misses # last-level cache, i.e. L3
 - l2_rqsts.references # všechny požadavky na L2
 - l2_rqsts.miss # L2 misses
 - longest_lat_cache.reference # všechny požadavky na L3
 - longest_lat_cache.miss # L3 misses
- Počet cyklů, kdy CPU pipeline stála a na něco čekala
cycle_activity.stalls_total
- Počet cyklů, kdy CPU pipeline stála kvůli čekání na L1/L2/L3 cache
 - cycle_activity.stalls_l1d_miss
 - cycle_activity.stalls_l2_miss
 - cycle_activity.stalls_l3_miss

Cvičení 4.1. Jak vyrobit predikované množství cache missu?

Řešení: uděláme pole, kde prvky jsou stejně velké jako velikost cache line. Pak procházet po poli. Může ale vadit prefetch, pokud budeme procházet lineárně. Můžeme randomizovat, nebo použít multiplikativní/aditivní grupy.

Cvičení 4.2. Jak změřit velikost třeba L1 cache (pomoci perf)?

Řešení: uděláme pole, kde prvky jsou stejně velké jako velikost cache line. Pak budeme binárním vyhledáváním měnit velikost pole dokud neuvidíme velký skok v cache missech.

Cvičení 4.3. Jak změřit asociativitu třeba L1 cache (pomoci perf)?

Řešení: uděláme pole, kde prvky jsou stejně velké jako velikost cache line. Pak přidáme velké mezery, taky by pomohlo pořídit velkou stránku (Linux, 4 GB)

Cvičení 4.4. Jak změřit hodně krátkou událost (miněno takovou, že měření je řadově pomalejší)?

Řešení: zopakujeme vícekrát (řadově miliony) ve smyčce. Problém ale je, že cykly jsou drahé, podobně volání funkce. Takže ideálně chceme rozbalit smyčky, vypnout optimalizaci a inlinovat.

Poznámka 4.5. Jak perf měří když OS se přepíná mezi contexty? (context switch)

Stejně jako při volání procedury, Linux uloží hodnoty flag registrů při context switchu.