Optimalizace programů

Dobrý, lepší, nejlepší (2)

V prvním dílu povídání o optimalizaci jsme se seznámili s několika technikami, které mohou vést ke zrychlení nebo ke zmenšení rozsahu přeloženého programu. Dnes toto povídání dokončíme. Připomeňme nejprve, že jsme si možné optimalizační postupy rozdělili do tří základních kategorií. V minulém čísle jsme probrali nejnižší a nejvyšší z nich a začali jsme se věnovat té nejrozsáhlejší – optimalizačním technikám, při nichž spolupracuje překladač. Nyní se podíváme na další možnosti.

Optimalizace cyklů

Cyklus by měl obsahovat jen příkazy, které je nezbytné opakovat – vše ostatní zdržuje. Jestliže např. chceme vyplnit všechny prvky pole a sinem x, můžeme napsat

```
for(int i = 0; i < N; i++) a[i] = sin(x);
```

Ovšem výpočet funkce sinus pro konstantní argument je zbytečné opakovat N-krát a slušný překladač tento cyklus upraví (říká se tomu *constant code elimination)* na tvar

```
pom = sin(x);
for(int i = 0; i < N; i++) a[i] = pom;
```

Přerovnávání instrukcí

Superskalární procesory, jako je Pentium a jeho následovníci, obsahují dvě výkonné jednotky. To znamená, že takový procesor může provádět dvě instrukce zároveň, ovšem za předpokladu, že na sobě nezávisí, tj. že výsledek jedné není vstupem druhé. Pokud na sobě instrukce závisí, bude jedna z výkonných jednotek blokována a program se tím zpomalí. Někdy je ovšem možné instrukce v přeloženém programu přerovnat *(instruction scheduling)* tak, aby k blokování nedocházelo. Podívejme se na učebnicový příklad. Jestliže přeložený program obsahuje instrukce

```
sub ax, 10 ; (1)
movsx ebx, ax ; (2)
xor ecx, ecx ; (3)
```

v uvedeném pořadí, pak první dvě instrukce na sobě závisí (v první odečítáme 10 od obsahu registru AX, ve druhé přesunujeme výsledek se znaménkovým rozšířením do registru EBX). Třetí instrukce nuluje obsah registru ECX a je na předchozích dvou nezávislá. Pokud by měl procesor provádět tyto instrukce v uvedeném pořadí, musel by s druhou instrukcí počkat, až skončí první. Jestliže je ale překladač přerovná do pořadí

```
sub ax, 10 ; (1)
xor ecx, ecx ; (3)
movsx ebx, ax ; (2)
```

mohou proběhnout instrukce (1) a (3) zároveň a k blokování nedojde.

Posun místo dělení a násobení

Celočíselné násobení mocninou dvou, tedy číslem tvaru 2^n, znamená vlastně jen posun o *n* bitů doleva, tj. výpočet x*8 je ekvivalentní výpočtu x << 3. Podobně dělení je ekvivalentní posunu doprava (se znaménkem). Náhrada aritmetických operací bitovým posunem vede nejen ke zrychlení, ale často i ke zkrácení programu. Jednak instrukce pro posun jsou samy o sobě výrazně rychlejší než aritmetické operace, jednak lze posuny provádět přímo v paměti, zatímco pro aritmetické operace musíme operandy nejprve přenést do registrů a pak výsledek zase uložit do paměti. Poznamenejme, že tuto optimalizaci *(strength reduction)* provádějí mnohé překladače automaticky, dokonce i při vypnutí všech optimalizačních přepínačů.

Slučování řetězců

Pokud se v programu opakují znakové řetězce, může je překladač sloučit, tj. použít ve všech případech odkaz na tutéž řetězcovou konstantu *(string pooling)*. To vede ke zmenšení programu, na rychlost to přímý vliv pochopitelně nemá.

Standardní rámec zásobníku

Při překladu podprogramů (tj. funkcí) se obvykle používá konstrukce, které se říká standardní

rámec zásobníku *(standard stack frame)*. Je to posloupnost instrukcí v úvodu a závěru funkce. Úvodní posloupnost, "prolog", definuje část zásobníku, která je pro aktuální volání funkce vyhrazena. K orientaci v této části paměti využívá překladač registru EBP, který obsahuje ukazatel na význačné místo v zásobníku, sloužící jako jakýsi "počátek soustavy souřadnic". Závěrečná posloupnost instrukcí, "epilog", tento rámec zruší a obnoví rámec funkce volající.

Jestliže překladači zakážeme používat standardní rámec zásobníku, bude jako vztažný bod používat místo EBP vrchol zásobníku, tedy registr ESP. Zjednoduší se tím o několik instrukcí tělo funkce (tedy zrychlí se nepatrně běh programu) a uvolní se tak registr EBP pro jiná použití (to může být podstatně významnější). Na druhé straně tím ale znesnadníme, nebo znemožníme činnost některým ladicím nástrojům, které na standardní rámec zásobníku spoléhají.

Podrobněji se na standardní rámec zásobníku někdy podíváme v samostatném článku.

Kontrola zásobníku

Volba umožňující zakázat, nebo povolit kontrolu zásobníku *(stack checking)* se vyskytovala už v 16bitových překladačích, její dnešní význam je však diametrálně odlišný. V obou případech zákaz kontroly zásobníku znamená zmenšení programu a zrychlení jeho běhu, ovšem za cenu jistého rizika běhové chyby.

V 16bitovém prostředí byla velikost zásobníku pevně stanovena při spuštění programu. Kontrola zásobníku tedy znamenala volání pomocné funkce, která při vstupu do podprogramu zjistila, zda je k dispozici dostatek místa pro jeho lokální proměnné. Teprve pak se v zásobníku potřebné místo vyhradilo.

Ve 32bitových Windows není kontrola tohoto typu nezbytná, neboť operační systém obsahuje mechanismus, který zabraňuje přetečení zásobníku. Abychom tento mechanismus pochopili, musíme si uvědomit, že Windows přidělují paměť programům po tzv. stránkách (zpravidla o velikosti 4 KB), a to ve dvou krocích. Nejprve paměť vyhradí *(reserve)*, pak ji předají *(commit)*. Přitom s pouze vyhrazenou pamětí program ještě nesmí pracovat – pokus o přístup do ní způsobí porušení ochrany paměti. Používat lze až paměť předanou.

Zásobník se skládá z několika stránek předaných programu. Poslední z nich přitom slouží jako jakési "hladinové čidlo": jestliže ji program použije, pochopí to operační systém jako upozornění, že může dojít k přetečení zásobníku, a předá programu další stránku.

Problémy zde mohou nastat, jestliže program deklaruje velké pole, které zabírá několik stránek. Deklarace

```
void test() {
int Pole[10000];
// ...
}
```

sice způsobí, že po vstupu do funkce *test()* se ukazatel na vrchol zásobníku posune nejméně o *sizeof(int)*10000* bajtů, ovšem tím se paměť programu ještě nepřidělí, pouze vyhradí. To znamená, že pokud doplníme předchozí ukázku do podoby

```
void test() {
int Pole[10000];
Pole[9999] = 999999; // Chyba?
}
```

může volání této funkce skončit porušením ochrany paměti, neboť deklarace proměnné *Pole* vyhradí téměř 10 stránek, které se ale programu nepředají; přitom však následující příkaz používá poslední z nich.

Standardně se proto po vyhrazení paměti volá pomocná funkce, která se "dotkne" po řadě všech nově vyhrazených stránek a tím způsobí jejich předání programu. (To je právě ona "kontrola zásobníku" ve Win32.)

Pokud si ovšem jsme jisti, že s prvky nově alokovaných velkých polí napoprvé budeme pracovat např. v pořadí podle rostoucích indexů, můžeme kontrolu zásobníku vypnout a tím zrychlit běh programu.

Překrývání proměnných v zásobníku

Občas se v jedné funkci vyskytnou vedle sebe lokální proměnné, jejichž "doby života" se nepřekrývají. To mohou být lokální proměnné deklarované ve dvou za sebou následujících blocích, mohou to ale také být lokální proměnné – řekněme x a y – deklarované ve stejném bloku, pro které platí, že poslední použití x předchází prvnímu použití y. Překladač může v takovém případě použít pro obě proměnné totéž místo v zásobníku *(stack overlays)*.

Jaký to může mít význam? Jednak se tím samozřejmě zmenšuje nebezpečí přetečení zásobníku, jednak to ale může vést i ke zmenšení a zrychlení běhu programu. Připomeňme si, že lokální proměnné jsou v programu adresovány relativně vzhledem k určitému význačnému bodu zásobníku, a to buď k místu, na které ukazuje registr EBP (jestliže používáme standardní rámec zásobníku), nebo k vrcholu zásobníku. Je-li lokální proměnná od tohoto význačného bodu vzdálena o méně než 128 bajtů, budou instrukce, které s ní pracují, o 3 bajty kratší než v případě, že bude od tohoto bodu vzdálena více.

Přezdívky

Jako "přezdívky" *(aliases)* se v této souvislosti označuje používání různých identifikátorů pro tutéž proměnnou. V C++ se za tím může skrývat především přístup k proměnné pomocí referencí nebo ukazatelů anebo použití proměnné jako složky anonymní unie; ve Fortranu by to mohlo být použití proměnné v příkazu EQUIVALENCE. Použijeme-li v programu přezdívky, znemožníme tím překladači celou řadu optimalizací, např. ukládání proměnných do registrů, šíření kopií nebo eliminaci společných podvýrazů, neboť překladač si nemůže být jist, zda se např. proměnné ve společném podvýrazu mezi jeho jednotlivými použitími nezmění, a zda ho tedy není třeba vyhodnocovat pokaždé znova.

Jestliže například napíšeme

```
a = x+y;
fun(&x);
b = x+y;
```

musí překladač předpokládat, že funkce *fun()* změní hodnotu proměnné x, a proto musí podvýraz vypočítat znovu.

V některých případech dokáže překladač zjistit, zda ukazatele nebo reference způsobí změnu proměnné, a v případě, že nikoli, výsledný program optimalizovat. To ovšem neplatí třeba v případě ukazatelů použitých jako skutečné parametry funkcí, neboť různé funkce mohou být překládány samostatně.

Ještě horší jsou ukazatele vracené funkcemi jako výsledek. Např. ukazatel *uk*, kterému přiřadíme hodnotu příkazem

```
char *uk = GetPtr();
```

může ukazovat na téměř kteroukoli proměnnou v programu a způsobovat tak její změny. Vyskytují-li se v programu podobné konstrukce, musí překladač použít konzervativní (pesimistickou) strategii a upustit od řady optimalizací. Volbou *assume no aliases* slibujeme překladači, že proměnné v programu nemají skryté přezdívky, a umožňujeme mu využít optimistickou strategii optimalizace.

Některé překladače, např. Visual C++, také nabízejí volbu *assume aliasing across function call*. Touto volbou říkáme, že přezdívky se v programu vyskytují pouze v rámci volání funkcí. To je sice velice vágní informace, ale i tak umožňuje alespoň některé optimalizace.

Sestavování jednotlivých funkcí

Zdrojový program může obsahovat funkce, které v programu nejsou vůbec volány. To mohou být např. nepoužité metody objektových typů z knihovny, mohou to ale být i funkce, které překladač použil jako vložené *(inline)*. Přeložené tělo funkce ovšem musí zůstat součástí souboru přeloženého programu (.OBJ), neboť při sestavování se může ukázat, že tuto funkci volá jiná část programu, překládaná samostatně. Optimalizaci tohoto druhu tedy může zajistit pouze sestavovací program *(linker)*.

Překladač ovšem může všechny funkce v přeloženém kódu "zabalit", tj. vložit do přeloženého modulu informace umožňující linkeru rozpoznat nepoužité funkce a při sestavování je eliminovat *(function-level linking)*. Tato optimalizace pochopitelně neovlivní přímo rychlost programu, může ale zmenšit jeho velikost.

Zdroje neefektivnosti

Předchozí odstavce ukazují, že většinu z "drobných" optimalizací by si – alespoň v principu – mohl udělat (dobrý) programátor sám. Není tedy optimalizace tohoto druhu zbytečná? Vždyť koneckonců programátor, který napíše (či spíše spáchá) cyklus

```
for(int i = 0; i < 100000; i++)
a[i] = sin(x);
```

by se měl vrátit do školy a zopakovat si základy programování.

Ve skutečnosti se ale s podobnými, a často i horšími konstrukcemi setkáme při rozvoji komplikovaných maker nebo šablon. Zdrojem neefektivnosti mohou být i dnes tak oblíbené generátory kódu (různí šamani, kteří na základě vyplněného dotazníku sestaví kostru aplikace, přidávají do aplikace funkce, třídy atd.), prostředky označované CASE, RAD atd., neboť tyto prostředky pracují na úrovni příliš vzdálené strojnímu kódu, než aby dokázaly generovat za všech okolností účinný program.

Dalším zdrojem neefektivit může být samotný překladač; koneckonců alternativní termín *kompilátor* vystihuje jeho činnost daleko přesněji: je to program, který na základě zdrojového textu **skládá** z předem připravených fragmentů cílový program, v pravém smyslu slova jej kompiluje. Výsledkem pak mohou být různé podivné konstrukce, které jsou sice věcně správné, ale nejsou optimální.

Například překladač Borland C++ 3.1, dodnes používaný pro vytváření dosových aplikací, umožňuje používat tzv. registrové pseudoproměnné, které programátorovi zpřístupňují obsah registrů procesoru. Příkaz

f(_FLAGS);ve kterém voláme funkci *f()* a jako parametr (typu unsigned) jí předáváme obsah registru příznaků, se přeloží posloupností

```
pushf
pop ax
push ax
call near ptr @f$qui
```

První dvě instrukce připraví obsah registru FLAGS do registru AX, třetí instrukce uloží takto připravenou hodnotu na vrchol zásobníku jako parametr funkce *f()*. Druhá a třetí instrukce je ovšem zbytečná; stačilo by

```
pushf
call near ptr @f$qui
```

Má to vůbec význam?

Z předchozího povídání by se mohlo zdát, že optimalizace na druhé a třetí úrovni nemá praktický význam. Koneckonců jde o jednotlivé instrukce, a my už víme, že zde se úspory pohybují v jednotkách nebo desítkách taktů. Ovšem jestliže se sečtou úspory z celého programu, který obsahuje mnoho cyklů, mohou být výsledky podstatné. Připojená tabulka ukazuje, jak se čas potřebný k setřídění pole lišil v případě, že překladač nepoužíval žádné optimalizace, a v případě, že použil standardní optimalizace pro konečnou verzi programu. (Jde o stejný program jako v minulém článku.)

Kdy a co optimalizovat

Je asi jasné, že optimalizace, kterou uživatel programu vůbec nezaznamená, je zbytečná. Rozdíl mezi tříděním haldou a vkládáním, který jsme ukázali v prvním dílu, je samozřejmě významný. Na druhé straně úspory vzniklé používáním registrových proměnných nemusí přesáhnout pár sekund na jeden běh programu, a tedy nemusí být nijak důležité.

Většina dnešních programů spolupracuje interaktivně s uživatelem a ten je zpravidla nejpomalejším článkem celého systému. To znamená, že – pokud jde o druhou úroveň optimalizace – je většinou vhodnější optimalizovat velikost programu, neboť té si uživatel všimne vždy. Optimalizace rychlosti má smysl především u numerických výpočtů, tedy u vědeckých a technických programů, a samozřejmě u časově kritických aplikací.

Miroslav Virius

Literatura

B. Zaratian: Microsoft Visual C++ 6.0 Programmer’s Guide. Microsoft Press 1998.
N. Wirth: Algoritmy a štruktúry údajov. Alfa, Bratislava 1988.