Robert Konieczny Grupa 1K232

Temat referatu: Omówienie technik optymalizacji kodu.

Optymalizacja kodu to proces mający na celu zmianę kodu przy zachowaniu obliczanej funkcji. Możemy przez to przyspieszyć działanie konkretnego programu, a przy okazji możemy zmniejszyć jego rozmiar. Optymalizacja polega na poprawianiu gotowego programu, gdyż często otrzymany kod nie jest optymalny. W procesie tym powinniśmy uwzględniać się wiele różnych kwestii. Najważniejszy jest czas działania docelowego programu. Zawsze istnieje możliwość znalezienia rozsądnego kompromisu pomiędzy potrzebą optymalizacji, a jej kosztem. Są jednak sytuacje kiedy nie warto tworzyć doskonale zoptymalizowanego kodu, ponieważ może to po prostu nie być opłacalne.

Kompilator może jedynie optymalizować kod w trzech kierunkach: aby działał on szybciej, albo aby zajmował mniej miejsca, albo jedno i drugie.

W trakcie optymalizacji poprawiamy efektywność kodu wynikowego, dokonujemy czasochłonnej analizy przepływu sterowania i przepływu danych. Optymalizacja musi zachować semantykę (poprawnego) programu, a efekt optymalizacji musi być zauważalny i wart wysiłku jak już wcześniej wspomniano.

Czasami zdarza się, że program przed i po optymalizacji daje różne rezultaty, najczęściej wynika to z błędów w programie. Powodem może być np. użycie niezainicjalizowanej zmiennej, błędnego odwołania z użyciem wskaźników itd.

Wygenerowanie programu tak aby odnaleziono miejsca, gdzie tracone jest najwięcej czasu. Zmiana algorytmu na poziomie kodu źródłowego czasami może dać najbardziej zauważalne rezultaty. Niektóre typy kompilatorów dokonują zaawansowanej analizy algorytmów i przetwarzają je na równoważne i wydajniejsze. Ma to szczególne znaczenie w przypadku generacji kodu dla maszyn równoległych i wektorowych. Optymalizacja może być wykonywana na poziomie kodu źródłowego. Dotyczy to profilowania programu, zmiany algorytmu czy przekształcenia pętli. Na poziomie reprezentacji pośredniej, gdzie dochodzi do ulepszenia pętli, wywołania procedur czy obliczanie adresów. Na etapie kodu wynikowego, w którym następuje przydział rejestrów, idiomy maszynowe oraz szeregowanie instrukcji. Optymalizacja może być lokalna na poziomie bloków podstawowych, a może też być globalna ponad blokami podstawowymi.

Stosując przekształcenia zachowujące funkcję programu możemy sprawnie optymalizować kod. Dobre rezultaty otrzymamy dzięki usuwaniu podwyrażeń wspólnych, propagację stałych, propagację kopii, usuwanie kodu martwego, zwijanie stałych, optymalizacji pętli przez przemieszczanie kodu, wyszukiwanie zmiennych indukcyjnych czy redukcję mocy w pętlach i wiele innych.

W tej części opracowania podane zostaną metody przekształceń nie zmieniających działania programu, a które są wykorzystywane do optymalizacji kodu. Dzięki takim zabiegom zyskujemy czas i ograniczamy jego wielkość. Jest to po prostu opłacalne dla zamawiającego napisanie programu jak i realizującego zlecenie.

Przykłady optymalizacji z pierwszej grupy:

1. **Wyliczanie wartości stałych**:
   jeśli to możliwe, obliczanie od razu wartości wyrażeń, w których występują stałe argumenty. Niektóre kompilatory potrafią także wyliczać w chwili kompilacji wartości funkcji dla stałych argumentów.

   ```
   x = 5 * 12 - 1;

   y = cos(0.0);
   ```

   może zostać uproszczone do

   ```
   x = 59;

   y = 1.0;.
   ```

2. **Propagacja stałych**:
   Jest to proces zastępowania wystąpień stałych przez ich wartości. Przykład

   x = 20;

   y = 3 * x;

   można zastąpić przez

   x = 20;

   y = 3 * 20;

   i dalej po zwinięciu wartości w drugim przypisaniu otrzymamy

   x = 20;

   y = 60;

   W wyniku propagacji stałych instrukcja definiująca stałą staje się martwym kodem,

   który można usunąć w trakcie eliminacji martwego kodu.

3. **Propagacja kopii:**

instrukcja x = y jest instrukcją kopiowania. Naturalne jest zastąpienie wystąpień zmiennej x przez y.

x = y;

a = x + y;

b = a + x;

Zastępujemy przez:

x = y;

a = y + y;

b = a + y;

Po wykonaniu propagacji kopii, instrukcja kopiująca jest martwym kodem. Ostatecznie otrzymamy:

a = y + y;

b = a + y;

4. **Upraszczanie wyrażeń logicznych i arytmetycznych** :

   zastosowanie znanych własności i tożsamości, np. unikanie mnożenia przez 1 czy dodawania 0.

5. **Eliminacja zbędnych wyrażeń**:

   pomijanie tych instrukcji lub wyrażeń, które nie są używane lub których obliczenie będzie miało ten sam wynik.

```
x = 7;

y = 5;

x = 12
```

   pierwsze przypisanie może zostać pominięte, bo wartość

   `x = 7;` nie jest używana i jedynie zastępowana nową wartością 12.

6. **Eliminacja wspólnych podwyrażeń:**

   gdy jakieś podwyrażenie występuje kilkakrotnie w różnych wyrażeniach, jest liczone tylko raz.

   x = 10 * i;

   y = 10 * i;

   z = x + y

Drugie wyliczenie 10 * i można zastąpić przez y. Przekształcony blok:

```
x = 10 * i;

y = x;

z = x + y
```

Eliminację wspólnych podwyrażeń można stosować lokalnie - gdy przeszukujemy jeden blok bazowy, globalnie - dla całego grafu przepływu (trudniejsze z powodu pętli).

7. **Łączenie instrukcji**:

   gdy kolejne instrukcje wykonują podobne działania są zastępowane jedną instrukcją,

```
x = x + 5;

x = x - 3;
```

   jest zastępowane jedną instrukcją

```
x = x + 2;
```

8. **Optymalizacja pętli przez przemieszczanie kodu:**

polega na zmniejszeniu ilości kodu w pętli, przez przemieszczenie obliczeń niezmienniczych przed pętle.

```
i = 0;
do {
  a = 13;
  i = i + 1;
} while (i < 15)
```

Wyrażeniem niezmienniczym jest instrukcja przypisania w trzeciej linii. Możemy (a wręcz należy) przemieścić ją przed pętlę.

9. **Wstawienie treści funkcji w miejscu wywołania**:

pozwala uniknąć kosztów wywołania podprogramu ma zastosowanie głównie dla krótkich funkcji, zawierających zwykle kilka instrukcji; jeśli funkcja jest wywoływana w programie raz, względnie niewielką liczbę razy również może zostać wstawiona w miejscu wywołania. Powoduje zwiększenie wielkości kodu wynikowego. Zyskujemy tutaj na pewno na wydajności, a najprawdopodobniej również na rozmiarze kodu. Na wydajności dlatego, że unikamy przygotowywania parametrów, wywołania i powrotu z podprogramu. Na rozmiarze, gdyż kod służący do przygotowania parametrów i wywołania funkcji raczej nie byłby

mniejszy niż sekwencja trzech instrukcji kodu maszynowego. Generalnie mamy tu do czynienia z poszukiwaniem kompromisu pomiędzy zyskiem prędkości wynikającym z nie wywoływania funkcji, a wzrostem rozmiaru programu wskutek wstawienia implementacji funkcji w miejsce jej wywołania.

10. **Usuwanie martwego kodu:**

zmienną nazywamy żywą w danym miejscu programu, jeżeli jej wartość może zostać użyta. Inaczej nazywamy ją w tym miejscu zmienną martwą. Analogicznie definiujemy kod martwy, tj. obliczający wartości bezużyteczne lub jest w kodzie źródłowym programu, który jest wykonywany, ale której wynik nigdy nie jest używany w żadnym innym obliczeniu. Realizacja martwego kodu marnuje czas obliczeń i pamięć.

11. **Łączenie pętli**:

    wykonywanie w jednej pętli instrukcji z sąsiednich pętli, mających te same warunki iterowania.

12. **Spłaszczanie pętli**:

    zastępowanie kilku zagnieżdżonych pętli jedną pętlą.

13. **Zwijanie stałych:**

Jest to proces upraszczania wyrażeń stałych.

x = 30 * 2;

można zastąpić przez

x = 60;

Zwijanie stałych czasem wykonuje się na wcześniejszych etapach kompilacji.

14. **Eliminacja identycznych argumentów funkcji**:

jeśli pewne argumenty funkcji są w każdym jej wywołaniu takie same, kompilator może zrezygnować z ich bezpośredniego przekazywania; np. jeśli te argumenty są stałe, zostaną wstawione wprost w treści funkcji.

15. **Eliminacja nieosiągalnego kodu**:

usuwanie z wyjścia nieosiągalnych fragmentów programu; ma na celu zmniejszenie rozmiaru kodu wynikowego.

16. **Zmiana kolejności bloków podstawowych:**

    w celu zmniejszenia liczby skoków.

17. **Łączenie bloków podstawowych**

18. **Eliminacja zbędnych instrukcji warunkowych**

19. **Optymalizacja przez dziurkę od klucza** lub **przez szparkę**:
    analiza wykonywana na niewielkiej liczbie sąsiednich instrukcji, której celem jest zastąpienie znanych wzorców instrukcji równoważnymi, specyficznymi dla danego procesora rozkazami pozwalającymi wykonać określone działania szybciej.
    Np. ustawienie lub wyzerowanie bitu można wykonywać standardowymi operacjami bitowymi (AND, OR), jednak istnieją procesory posiadające rozkazy, które wprost działają na bitach.

20. **Rozwijanie pętli**:

polega na powtórzeniu treści pętli kilka razy i jednocześnie wykonaniu proporcjonalnie mniejszej liczby obiegów pętli. Koszt wykonywania instrukcji skoku koniecznych w realizacji pętli, są w procesach potokowych znaczne. Ten rodzaj optymalizacji pozwala osiągnąć większą wydajność.

Optymalizacja wiąże się również z alokacją rejestrów, czyli określeniem, w których rejestrach procesora będą zapisywane wartości na poszczególnych etapach obliczeń. Dąży się do minimalizacji liczby przesłań wartości między rejestrami, a pamięcią, w szczególności często wykorzystywane wartości powinny znajdować się w rejestrach.

Kompilator może też optymalizować powszechnie używane funkcje biblioteczne . Wiedząc jak działają w pewnych przypadkach zastępuje wywołania funkcji równoważnym ciągiem instrukcji. Np. w języku C funkcja `memset` zeruje wskazany obszar pamięci, dla niewielkich obszarów kompilator może wstawić kilka instrukcji procesora zerujących pamięć.

Podobnie, zamiast wywołania funkcji formatującej `printf("%s\n", napis)`, która w tym przypadku jedynie wypisze napis, można zastosować szybszą funkcję `puts(napis)`.