Maciej Czekaj

Kod Schrödingera

czyli maszyna abstrakcyjna języka C

Język C uważany jest za niskopoziomowy i „bliski sprzętowi". Tymczasem model języka C opiera się na abstrakcyjnej maszynie, która z dużą swobodą realizuje program, często działając wbrew potocznym intuicjom na temat kodu. Abstrakcyjna maszyna C „materializuje się" w postaci efektów ubocznych programu, przypominając tym samym słynny paradoks kota Schrödingera, który tym bardziej jest w pudełku, im bardziej tam zaglądamy.

ZASADA NIEOZNACZONOŚCI PROGRAMÓW

Standard ISO C11 w rozdziale dotyczącym wykonania programu (5.1.2.3 Program Execution) wprowadza pojęcie maszyny abstrakcyjnej, która podczas wykonania programu generuje efekty uboczne (ang. *side effects*). Efektem ubocznym może być:

- zapis do zmiennej w pamięci,
- dostęp do zmiennej `volatile`,
- operacja wejścia/wyjścia,
- wywołanie funkcji, która wykonuje jedno z powyższych zadań.

Dodatkowo efekty uboczne są w pełni widoczne tylko w niektórych „punktach" programu, zwanych „punktami sekwencji" (ang. *sequence points*). Przekładając zawiły, prawniczy język standardu C na ludzką mowę, oznacza to, że program w C realizuje się w sposób dowolny, o ile nie jest „zmuszony" wywołać efektów ubocznych, np. drukując coś na ekran (czyli operacją we/wy). Takie zachowanie programu w C można roboczo nazwać „Zasadą Nieoznaczoności Programu". Czy ta zasada ma jakieś implikacje dla programistów? Na to pytanie spróbujemy odpowiedzieć w niniejszym artykule.

Zagadnienia poruszane poniżej są omawiane na przykładzie języka C, ale w równym stopniu dotyczą języka C++, a także częściowo mogą aplikować się do innych imperatywnych języków kompilowanych, jak D, Go, Pascal itd.

Programy będą omawiane na przykładzie kodu asemblera generowanego przez kompilator C GCC 4.8 lub Clang 3.6. Oba generują podobny kod, ale użycie Clanga pozwala na wyabstrahowanie analizy kodu od architektury procesora, gdyż Clang może produkować tzw. kod pośredni, zbliżony do asemblera, ale bardziej czytelny dla programisty (GCC też to potrafi za pomocą rodziny opcji `-fdump-tree`).

POLOWANIE NA EFEKTY UBOCZNE

Jak wspomniano wyżej, efekty uboczne „ujawniają się" tylko w sytuacjach, kiedy program komunikuje się ze światem zewnętrznym. Istnienie efektów ubocznych jest zarazem podstawą programowania imperatywnego, jak i jego największą bolączką. Efekty uboczne w znaczący sposób utrudniają kompilatorowi analizę programów i – w konsekwencji – ich optymalizację.

Listing 1.1. Kod źródłowy funkcji code_1a

```
void use(int var);

void code_1a() {
  int a,b;
  a = 1;
  b = a + 2;
  printf("Hello");
  a = b + 1;
  use(a);
  b = a + 1;
  use(b);
}
```

Listing 1.2. Kod maszynowy skompilowanej przez GCC (x86_64) funkcji code_1a

```
code_1a:
sub    rsp,0x8   # ramka stosu, wymóg ABI
mov    esi,0x4008a4
mov    edi,0x1
xor    eax,eax
call   4004a0 <__printf_chk@plt>
mov    edi,0x4   # edi = 4, 1-szy argument funkcji
call   4007e0 <use>  # use(4)
mov    edi,0x5 # edi = 5
add    rsp,0x8
jmp    4007e0 <use>  # use(5)
nop    DWORD PTR [rax]
```

Listing 1.3. Kod wynikowy LLVM funkcji code1_a

```
define void @code_1a() #0 {
  %1 = tail call i32 (i8*, ...)* @printf <...>
  tail call void @use(i32 4) #4
  tail call void @use(i32 5) #4
  ret void
}
```

Listingi 1.1, 1.2, 1.3 pokazują tę samą funkcję w trzech różnych postaciach:

- kod źródłowy
- skompilowaną przez gcc i zdeasemblowaną:

```
gcc -O3 -fno-tree-vectorize -fno-unroll-loops -c -o kod.o kod.c
objdump -d kod.o > kod.s
```

- skompilowaną przez clang komendą:

```
clang-3.6 -O3  -fno-unroll-loops -fno-reroll-loops -fno-vectorize
-S -emit-llvm kod.c
```

Ten sam schemat kompilacji zostanie zastosowany w dalszych listingach, ale już bez GCC. Opcje `-fno-unroll-loops` itp. zostaną objaśnione później. W pierwszym przykładzie GCC ma służyć do porównania między kodem asemblera a kodem pośrednim LLVM. Plik `kod.c` zawiera omawiane przykłady.

W programie na Listingu 1.1 występują dwie zmienne a i b, które są zmiennymi lokalnymi. Wszystkie zmienne lokalne alokowane są na stosie

SEMIHALF
EMBEDDED SYSTEMS

Wypłyń na nieznane akweny kodu i nawiguj do celu tak, jak chcesz

Jeśli masz minimum 2 lata doświadczenia na stanowisku programistycznym, płynnie posługujesz się językiem C, pasjonują cię systemy operacyjne i sieci, aplikuj na stanowiska:

- **Software Engineer**
- **Senior Software Engineer**

Potrzebujesz więcej informacji?

telefon: +48 12 296 43 35
email: jobs@semihalf.com
web: www.semihalf.com/jobs.html

Rekrutację prowadzi Wojciech Szymański

www.semihalf.com | jobs@semihalf.com

funkcji, ale czy muszą być? Na Listingu 1.2 rzuca się w oczy brak alokacji miejsca na stosie dla zmiennych, bo rejestr stosu *rsp* jest zmniejszany o 8 bajtów, ale nic na stosie nie jest zapisywane. Co więcej, wywołanie funkcji `use(int)` odbywa się ze stałą, a nie zmienną! Rejestr *edi* (*rdi*), według System V ABI (standardu binarnego języka C na Unix-y) na x86_64, to pierwszy argument funkcji (drugim jest *rsi*) i jest on ładowany stałą 4 przed pierwszym wywołaniem funkcji `use()`, a potem wartością 5.

Listing 1.3 rozjaśnia nieco sytuację, usuwając zawiłości asemblera. Kod LLVM wyraźnie pokazuje użycie stałych 4 i 5 przy wywołaniu funkcji `use()`. Wyrażenie „tail call" zastępuje tu całą procedurę wywołania funkcji, a poza tymi wyrażeniami nie ma nic. Żadnych zmiennych! Żadnych operacji arytmetycznych! Kompilator sam wykonał wszystkie działania i następnie użył ich wyniki w miejsce zmiennych. Co decyduje o tym, czy takie „pójście na skróty" jest legalne?

Zanim zaczniemy wysuwać wnioski, zmodyfikujmy nieco funkcję, tak aby użyty został adres zmiennej a.

Listing 2.1. Kod źrodłowy funkcji code_1b

```
void escape(int *var);

void code_1b() {
  int a,b;
  a = 1;
  b = a + 2;
  printf("Hello");
  a = b + 1;
  escape(&a);
  b = a + 1;
  use(b);
}
```

Listing 2.2. Kod wynikowy funkcji code_1b

```
define void @code_1b() #0 {
  %a = alloca i32, align 4 ; rezerwuj stos dla zmiennej a
  %1 = tail call i32 (i8*, ...)* @printf( <...>
  store i32 4, i32* %a, align 4, !tbaa !1 ; a = 4
  call void @escape(i32* %a) #4
  %2 = load i32* %a, align 4, !tbaa !1 ; ponownie wczytaj a
  %3 = add nsw i32 %2, 1               ; a += 1
  call void @use(i32 %3) #4
  ret void
}
```

Funkcja `code_1b()` różni się od poprzedniczki tym, że teraz zmienna a jest przekazywana do funkcji `escape(int &)` przez wskaźnik, a wcześniej była przez wartość do funkcji `use(int)`. Kod wynikowy LLVM nagle „spuchł" i pojawiło się kilka nowych wyrażeń:

- alloca – rezerwacja miejsca na stosie
- store, load – zapis/odczyt z pamięci
- add – operacja arytmetyczna

Wreszcie widać dodawanie! Nadal jednak jest tylko jedno i pojawia się tuż po wywołaniu funkcji `escape()`. Spróbujmy zatem jeszcze bardziej skomplikować sytuację:

Listing 3.1. Kod źródłowy funkcji code_1c

```
int a,b;

void code_1c() {
  a = 1;
  b = a + 2;
  printf("Hello");
  a = b + 1;
  use(a);
  b = a + 1;
  use(b);
}
```

Listing 3.2. Kod wynikowy funkcji code_1c

```
define void @code_1c() #0 {
  store i32 1, i32* @a, align 4, !tbaa !1    ; zapisz a = 1
  store i32 3, i32* @b, align 4, !tbaa !1    ; zapisz b = 3
  %1 = tail call i32 (i8*, ...)* @printf(i8* getelementptr
  inbounds ([6 x i8]* @.str, i64 0, i64 0)) #4
  %2 = load i32* @b, align 4, !tbaa !1       ; wczytaj b
  %3 = add nsw i32 %2, 1                     ; b += 1
  store i32 %3, i32* @a, align 4, !tbaa !1   ; zapisz a
  tail call void @use(i32 %3) #4 ; use(b)
  %4 = load i32* @a, align 4, !tbaa !1       ; wczytaj a
  %5 = add nsw i32 %4, 1                     ; a += 1
  store i32 %5, i32* @b, align 4, !tbaa !1   ; zapisz b
  tail call void @use(i32 %5) #4             ; use(b)
  ret void
}
```

Listing 3.1 to funkcja z Listingu 1.1, ale tym razem zmienne a i b są globalne. Kod wynikowy na listingu 3.2 jest znów dłuższy, a operacji arytmetycznych jest więcej.

Zmienne a i b otrzymują wartości początkowe 1 i 3, czyli kompilator uprościł wyrażenie `b = a + 2` do `b = 3`. Wszelkie pozostałe operacje są jednak wykonywane bez „oszustw", czyli np. wyrażenie `a = b + 1` jest realizowane przez:

- załadowanie wartości b z pamięci do rejestru (zmiennej tymczasowej)
- dodanie 1 do rejestru
- zapisanie rejestru do zmiennej a w pamięci

Cała powyższa seria listingów obrazuje, jak kompilator wnioskuje na temat efektów ubocznych, dzieląc je na konieczne i te, które można pominąć. Ta działalność kompilatora nosi nazwę *escape analysis* i opiera się na kilku prawach o „wyciekaniu" zmiennych. Na czym polega wyciek zmiennej?

Wyciekanie adresu zmiennej to sytuacja, gdy w programie istnieje kilka kopii adresu danej zmiennej i nie znajdują się one w zasięgu jednej funkcji.

Inaczej mówiąc, wyciekająca zmienna zachowuje się tak, jak zmienna globalna, bo jej adres jest używany przez wiele funkcji w programie.

Zmienna wycieka, gdy:

- jest zmienną globalną, bo można jej użyć w każdej funkcji,
- jej adres jest przekazany do innej funkcji, której definicja nie jest znana (nie wiadomo, jak działa),
- jej adres jest skopiowany do innej zmiennej, która wycieka (np. globalnej).

Zmienna nie wycieka, gdy:

- jest zmienną lokalną, której adres nie jest znany poza blokiem funkcji,
- jest zmienną tymczasową (rvalue), czyli przejściowym etapem obliczeń.

Zmienna niewyciekająca ma specjalne własności:

- jej użycie w operacjach arytmetycznych i logicznych nie daje efektów ubocznych,
- może nie zajmować żadnej pamięci (ale nie musi, np lokalna tablica lub struktura będzie miała pamięć na stosie, bo jest za duża na trzymanie w rejestrach),
- można ją uprościć lub wyeliminować, zachowując semantykę programu (patrz przykład: `a = 1; b = a + 1 => a = 1; b = 2;`),
- jej wartość może być przechowywana w innej zmiennej tymczasowej (np w rejestrze), bo wiadomo, że jej wartość w pamięci się nie zmieni poza bieżącym blokiem kodu.

Zmienna wyciekająca natomiast:

- użyta, daje efekty uboczne (odczyt lub zapis do pamięci),
- musi mieć zarezerwowane miejsce w pamięci (stos lub segment globalny),