Optymalizacje na wyciągnięcie ręki

Kompilatory już dawno przestały być jedynie translatorami kodu źródłowego na kod maszynowy. Obecnie oferują programiście coraz bardziej wyrafinowane możliwości analizy i optymalizacji. Warto z nich skorzystać, zanim przystąpi się do żmudnego i podatnego na błędy, ręcznego usprawniania napisanego wcześniej kodu.

Kompilatory są podstawowymi narzędziami, jakie wykorzystuje programista w swojej codziennej pracy. Pozwalają one nie tylko na generowanie kodu wynikowego w oparciu o kod źródłowy, ale także, a może przede wszystkim, na jego optymalizowanie. Standardowe procedury optymalizacyjne są załączane za pomocą flagi `-O<arg>`, gdzie parametr `arg` opisuje zbiór predefiniowanych operacji, jakie kompilator wykonuje, aby wygenerować bardziej wydajny (lub o mniejszym rozmiarze) kod wynikowy. Zazwyczaj, gdy kod wygenerowany przez kompilator nie spełnia wymagań wydajnościowych, programiści przystępują do czasochłonnych, ręcznych optymalizacji. Twórcy kompilatorów wychodzą naprzeciw rosnącym oczekiwaniom programistów i z każdą kolejną wersją kompilatora dostarczają coraz bardziej wydajne możliwości optymalizacji. Pozwala to na ominięcie kłopotliwego ręcznego dostrajania kodu. Celem tego artykułu jest zaprezentowanie na przykładzie projektu LLVM [5] i kompilatora Clang dodatkowych optymalizacji, które pozwolą wygenerować wydajniejszy kod bez potrzeby modyfikacji źródeł.

ORGANIZACJA PROCESU KOMPILACJI

Zanim zostaną omówione następujące optymalizacje: czasu linkowania (ang. *Link Time Optimization* – LTO) oraz opierająca się na danych z profilowania kodu (ang. *Profile Guided Optimization* – PGO), warto przedstawić pokrótce podstawowe informacje dotyczące budowy kompilatorów i samego procesu kompilacji. Pozwoli to zrozumieć, w jaki sposób wymienione wyżej optymalizacje umożliwiają wygenerowanie bardziej wydajnego kodu. Dokładny opis projektu Clang/LLVM można znaleźć w numerze 08/2014 „Programisty".

Obecnie oferowane kompilatory wspierają zarówno wiele języków programowania, jak i liczne architektury docelowe. Jest to możliwe dzięki wprowadzeniu wspólnej reprezentacji pośredniej (ang. *Intermediate Representation* – IR) [4]. Kod wejściowy napisany przez programistę jest dzielony na jednostki translacji (ang. *Translation Unit*) [14]. Każda jednostka translacji zawiera kod z pliku *.c/*.cpp (wraz z dołączonymi nagłówkami) i na jej podstawie generowany jest kod IR. Stanowi on podstawę do przeprowadzenia wymaganych etapów optymalizacji, które są niezależne od docelowej architektury. Dzięki tym optymalizacjom możliwe jest m.in. uproszczenie pętli, zastąpienie wyrażeń matematycznych stałą wartością czy wyeliminowanie fragmentów kodu, które nigdy nie zostaną wykonane. Warto zaznaczyć, że kompilator generuje i optymalizuje kod IR dla każdej jednostki translacji niezależnie od sposobu wykorzystania zaimplementowanego kodu w pozostałych częściach programu. Kompilator na tym etapie nie ma wiedzy, czy i jak często funkcje zaimplementowane w danej jednostce translacji są wykorzystywane w całym projekcie. Po skończonej optymalizacji kodu IR generowany jest kod maszynowy odpowiedni dla danej platformy docelowej. Pliki z kodem maszynowym stanowią podstawę dla procesu linkowania, który polega na łączeniu plików binarnych w działającą aplikację. Zaletą takiej organizacji procesu kompilacji jest możliwość zrównoleglenia przetwarzania plików źródłowych na pliki binarne. Ponadto modyfikacja pojedynczego pliku źródłowego wiąże się jedynie z ponownym wygenerowaniem odpowiedniego pliku binarnego i uruchomieniem linkera.

LTO – ROZSZERZENIE MOŻLIWOŚCI OPTYMALIZACYJNYCH

Standardowy proces kompilacji opisany powyżej uniemożliwia kompilatorowi przeanalizowanie i zoptymalizowanie zależności pomiędzy funkcjami zdefiniowanymi w różnych jednostkach translacji. Powyższy problem zilustrowano w Listingu 1.

Listing 1. Przykład kodu zaimplementowanego w kilku jednostkach translacji

```
---shared.h---
int foo();
int foo1();
int bar();

---foo.c---
#include "shared.h"

int foo1() { return 1; }

int foo() {
     int res = 0;
     for (int i = 0; i < 10; ++i)
          res += bar();
     return res;
}

---bar.c---
#include "shared.h"

int bar() { return foo1(); }

---main.c---
#include "shared.h"
#include <stdio.h>

int main() {
     int res = foo();
     printf("Foo: %d", res);
     return 0;
}
```

Analizując osobno poszczególne pliki źródłowe, nie można przewidzieć, w jaki sposób funkcje zdefiniowane w danym pliku są wykorzystywane w innych. Gdyby implementacja funkcji `bar` została przesunięta do pliku *foo.c*, to kompilator mógłby przeprowadzić optymalizacje zilustrowane w Listingu 2. W zaprezentowanym przykładzie z Listingu 1 kompilator generując kod wynikowy dla pliku *foo.c*, nie może uprościć pętli `for` w funkcji `foo`, ponieważ nie wie,