NOKIA

Praktyczne aspekty inżynierii oprogramowania

Optymalizacje w C++

- Zbigniew Borkowski & Marcin Grzebieluch

- 04-05-2016

Agenda

- Czy istnieje kod optymalny?
- Kiedy optymalizować?
- Metody pomiarów wydajności
- Problem przedwczesnej optymalizacji
- Optymalizacje kompilatora
- Optymalizacje CPU
- Tips & tricks
- Podsumowanie

Kod optymalny

Nie ma czegoś takiego jak kod optymalny!

Kod optymalny

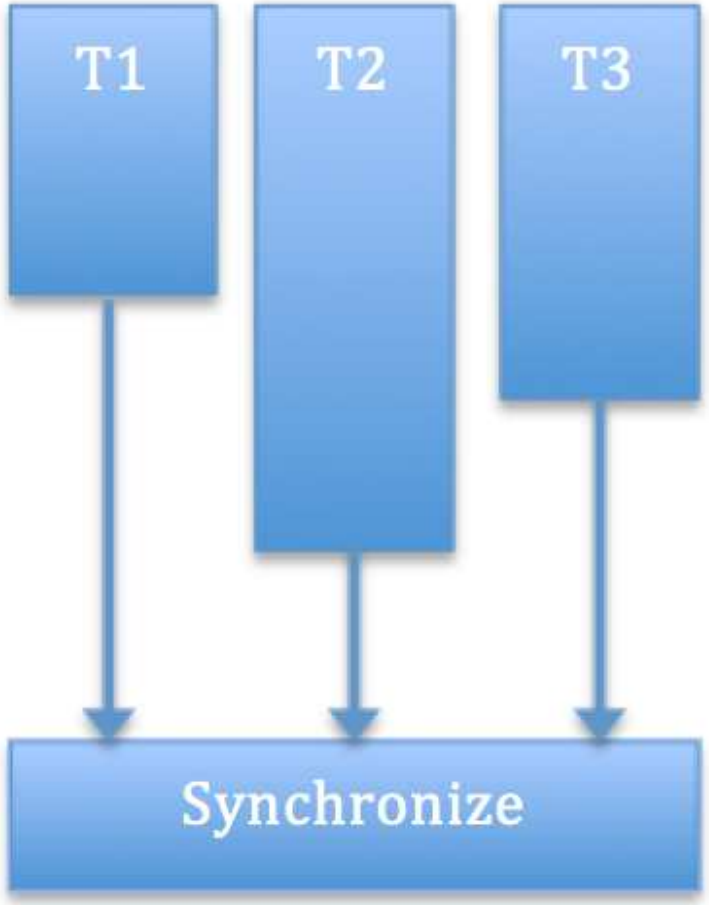
- **Kod może być zoptymalizowany pod jakimś względem:**
  - zużycia czasu procesora
  - pamięci
  - zapisów na dysku
  - czasu wykonania
  - wielkości pliku binarnego

Kiedy optymalizować kod?

- Gdy nasz program działa bezbłędnie!
- Gdy wyniki testów pokazują, że istnieje taka potrzeba
- Gdy dokładnie wiemy co jest wąskim gardłem naszej aplikacji

Co dokładnie optymalizować?

- T1, T2, T3 to wątki synchronizowane w jednym punkcie
- Optymalizacja w ciemno T1 lub T3 nie da nam nic pod względem czasu wykonania aplikacji



Prawo Amdahla

Wpływ wydajności komponentu na wydajność systemu jest wprost proporcjonalny do udziału tego komponentu w systemie.

“Optimization matters *only* when it matters.”

Poleganie na intuicji – który kod jest szybszy?

```
int numOfUnique(vector<int> const & v)
{
  auto s = std::set(begin(v), end(v));
  return s.size();
}
```

```
int numOfUnique(vector<int> v)
{
  sort(begin(v), end(v));
  v.erase(unique(begin(v), end(v)),
     end(v));
  return v.size();
}
```

Skąd wiadomo co optymalizować?

- Dzisiejsze komputery są zbyt skomplikowane, żeby polegać na intuicji
- Kompilatory robią bardzo dużo, żeby kod wynikowy nie przypominał tego co napisaliśmy
- Człowiek ma tendencję do upraszczania modelu i bazowaniu na błędnych założeniach
- Zawsze należy mierzyć

Jak mierzyć?

```
int main()
{
     startMeasurment();
     for(int i = 0; i < X; ++i)
     {
          //Our code
     }
     reportMeasurment();
     useResultOfOurCode();
}
```

Jak mierzyć?

```
int main()
{
     for(int i = 0; i < X; ++i)
   {
          startMeasurement();
          //our code
          reportMeasurement();
          useResult();
   }
     agregateMeasurements();
}
```

Co zrobić by kompilator nie usunął nam wyników?

- `volatile auto = result;`
- `std::cout << result;`

Jak mierzyć?

- $tm = t + tq + tn + to$
- t – czas algorytmu
- tq – szum kwandyzacji
- tn – szum z innych źródeł
- to – narzut spowodowany pomiarem
- $tq + tn + to$ – daje nam szum prawie gausowski (nie może być ujemny)
- Dla wielu pomiarów tego samego kodu t będzie stałe

Czym mierzyć

- `std::chrono::high_resolution_clock`
  - Może nie być stabilny (można sprawdzić za pomocą `is_steady`)
- Linux api: `clock_gettime`

Zmierzmy co jest szybsze

```
int numOfUnique(vector<int> const & v)
{
  auto s = set<int>(begin(v),end(v));
  return s.size();
}
```

```
int numOfUnique(vector<int> v)
{
  sort(begin(v), end(v));
  v.erase(unique(begin(v),
     end(v)), end(v));
  return v.size();
}
```

Punkt odniesienia

“Posortowałem milion floatów w 5ms”

Punkt odniesienia

- Przed rozpoczęciem optymalizacji należy mieć punkt odniesienia
- Punkt odniesienia musi być zmierzony w takich samych warunkach co nasza optymalizacja.
- Dobrym punktem odniesienia jest wcześniejsza implementacja lub biblioteka standardowa

Wykrywanie problemów wydajnościowych

- Podstawowym narzędziem do definiowania problemów wydajnościowych są jasno sprecyzowane wymagania na temat wydajności (np. 100 zapytań http na sekundę)
- Jeżeli wymagań nie ma, lub są one nieprzemyślane, może dojść do absurdów
- Problemy wydajnościowe w aplikacjach rozproszonych często mogą objawiać się nie wprost

Profilowanie

Wyróżniamy 3 typy profilowania:

- Profilowanie zdarzeń (Event based profiling) – alokacje, zawołania itd.
- Profilowanie statystyczne – sprawdzanie stosu z określoną częstotliwością
- Instrumentalizacja kodu – wprowadzenie do programu dodatkowych funkcji odpowiedzialnych za zbieranie statystyk

Profilowanie zdarzeniowe

- Nie wymaga ingerencji w binarkę*
- Wymaga środowiska które zapewni zliczanie zdarzeń
- Dodatkowe środowisko znacząco wpływa na czas wykonania
- Najpopularniejszy profiler zdarzeniowy dla C i C++: valgrind

Valgrind

- Zestaw narzędzi służących do debugowania i profilowania aplikacji odpalanych w piaskownicy
- Najpopularniejsze narzędzie do sprawdzania wycieków pamięci
- Zapewnia narzędzia do profilowania użycia pamięci cache, profilowania wywołań funkcji, błędów wątków, zużycia pamięci dynamicznej

Przykładowe użycie valgrinda

- `$valgrind ls -a`
- `$valgrind -tool=callgrind ls -a`
- W przypadku użycia callgrinda powstanie plik callgrind.out.[PID], którego możemy użyć w narzędziach do interpretacji takich jak callgrind_anotate czy gprof2dot

Zadanie

- Za pomocą valgrinda sprawdź, co ile wartości `std::vector` jest przealokowywany
- Ogranicz liczbę alokacji do jednej
- Zmierz czas przed i po optymalizacji

Profilowanie statystyczne

- Nie wymaga ingerencji w binarkę*
- Opiera się na próbkowaniu procesu z zadaną częstotliwością
- Możliwe rozpoczęcie profilowania chodzącego już programu
- Najpopularniejszy program do profilowania statystycznego w linuksie: `perf`
- Specyficznym przykładem profilu statystycznego jest narzędzie `top`

perf

- Zapewniony w paczce linux-tools
- Podstawowe narzędzie do analizy wydajności systemu linux
- Dostępny od wersji jądra > 2.6

Instrumentalizacja programu

- Polega na dodaniu dodatkowego kodu do naszego programu odpowiedzialnego za wytworzenie danych
- Wprowadza znaczący narzut na czas działania programu
- Wymaga przebudowania naszego programu na potrzeby profilowania
- Program musi zamknąć się poprawnie, aby zwrócić dane

Użycie gprofa

- `$g++ -g -pg test.cpp`
- `$./a.out`
- Powstanie plik gmon.out, który można zinterpretować za pomocą perfa i binarki a.out
- `$perf > profling.txt`

gprof2dot

- Pozwala na przekonwertowanie wyników profilerów: calgrinda, gprofa, perfa i wielu innych na czytelną formę graficzą
- https://github.com/jrfonseca/gprof2dot

“We should forget about small efficiencies, say about 97% of the time. Premature optimization is the root of all evil”

Donald E. Knuth

Problem przedwczesnej optymalizacji

- Zasada Pareta - 80/20
- Czytelność vs wydajność
- Pomiary!
- Jakość tworzonego rozwiązania
- Algorytmy
- Architektura

Czy komputery wykonują programy, które piszemy?

Czynniki wpływające na optymalizacje kompilatora

- Architektura CPU (RISC vs CISC)
- Rozszerzenia (MMX, SSE, 3DNow!)
- Liczba rejestrów procesora
- Liczba jednostek wykonawczych
- Rozmiar cache'u

Optymalizacje kompilatora nie współgrają z trybem debugowym

Optymalizacje kompilatora

- Minimalizacja liczby wykonywanych operacji
- Zastąpienie kosztownych operacji prostszymi
- Minimalizacja chybień cache'u (cache miss)

- Minimalizacja rozmiaru pliku wykonywalnego

- Minimalizacja ilości zużywanej energii

Łączenie instrukcji

```
int i;

void f (void)
{
  i++;
  i++;
}
```

```
int i;

void f (void)
{
  i += 2;
}
```

Zwijanie stałych

```
int meaningOfLife()
{
  return 40 + 2;
}
```

```
int meaningOfLife()
{
  return 42;
}
```

Propagacja stałych

```
int x = 5;
int y = x + 6;
```

```
int x = 5;
int y = 11;
```

Eliminacja wspólnych podwyrażeń

```
void foo (int x, int y)
{
  int i = x + y + 1;
  int j = x + y;
  return i * j;
}
```

```
void foo (int x, int y)
{
  int tmp = x + y;
  int i = tmp + 1;
  int j = tmp;
  return i * j;
}
```

Eliminacja martwego kodu

```
int global;
void foo ()
{
  int i;
  i = 1;
  global = 1;
  global = 2;
  return;
  global = 3;
}
```

```
int global;
void foo ()
{
  global = 2;
  return;
}
```

Zamiana dzielenia/mnożenia na przesunięcie bitowe

```
int foo (unsigned int i)
{
  return i / 2;
}
```

```
int foo (unsigned int i)
{
  return i >> 1;
}
```

Zwijanie pętli

```
int m[100][200];

for (i = 0; i < 200; i++)
{
  for (j = 0; j < 100; j++)
  {
    m[j][i] = 0;
  }
}
```

```
int m[100][200];
int *ptr = &m[0][0];

for (i = 0; i < 20000; i++)
{
  *ptr++ = 0;
}
```

Łączenie pętli

```
for (i = 0; i < 100; i++)
  m1[i] = m1[i] + 10;

for (i = 0; i < 100; i++)
  m2[i] = m2[i] + 20;
```

```
for (i = 0; i < 100; i++)
{
  m1[i] = m1[i] + 10;
  m2[i] = m2[i] + 20;
}
```

Wyciągnięcie wyrażenia z pętli

```
int m[100];

void foo (int x, int y)
{
  int i;

  for (i = 0; i < 100; i++)
  {
    m[i] = x + y;
  }
}
```

```
int m[100];

void foo (int x, int y)
{
  int i;
  int tmp = x + y;

  for (i = 0; i < 100; i++)
  {
    m[i] = tmp;
  }
}
```

Rozwinięcie pętli

```
for (i = 0; i < 100; i++)
  g ();
```

```
for (i = 0; i < 100; i += 2)
{
  g ();
  g ();
}
```

Uproszczenie wyrażeń warunkowych

```
void foo (void *ptr)
{
  if (ptr != NULL)
  {
    bar(1);
    if (ptr) bar(2);
    bar(3);
  }
}
```

```
void foo (void *ptr)
{
  if (ptr != NULL)
  {
    bar(1);
    bar(2);
    bar(3);
  }
}
```

Wyciągnięcie wyrażenia warunkowego

```
for (int i = 0; i < 100; i++)
  if (x)
    a[i] = 0;
  else
    b[i] = 0;
```

```
if (x)
  for (int i = 0; i < 100; i++)
    a[i] = 0;
else
  for (int i = 0; i < 100; i++)
    b[i] = 0;
```

Wstawianie funkcji (inline)

```
int add (int x, int y)
{
  return x + y;
}

int sub (int x, int y)
{
  return add (x, -y);
}
```

```
int sub (int x, int y)
{
  return x - y;
}
```

Usunięcie zmiennych indukcyjnych

```
int a[100];
int b[100];

void foo ()
{
  int i1, i2, i3;

  for (i1 = 0, i2 = 0, i3 = 0;
       i1 < 100;
       i1++)
    a[i2++] = b[i3++];
}
```

```
int a[100];
int b[100];

void foo ()
{
  int i1;

  for (i1 = 0; i1 < 100; i1++)
    a[i1] = b[i1];
}
```

Usunięcie wywołania funkcji wirtualnej

```
void drawCircle(uint32_t radius)
{
  Shape* obj = new Circle(radius);
  obj->draw(); //dynamic binding
}
```

```
void drawCircle(uint32_t radius)
{
  Circle* obj = new Circle(radius);
  obj->draw(); //static binding
}
```

Optymalizacja wartości zwracanej (RVO)

```
BigClass createBig ()
{
  BigClass b(1000);
  prepareMagic(&b);
  return b;
}

BigClass obj = createBig();
```

```
void createBig (BigClass* obj)
{
  obj->BigClass(1000); //constructor
  prepareMagic(obj);
}

BigClass obj; //constructor not executed here
              //just stack memory reservation
createBig(&obj);
```

CPU też wprowadza optymalizacje

- Potoki (Pipelines)
- Superpotoki
- Superskalarność
- Przewidywanie rozgałęzień (branch prediction)

Tips & tricks

- Popranie inicjalizuj obiekty:

```
X x;
x = 10;
```

```
X x(10);
```

Tips & tricks

- Minimalizuj liczbę obiektów tymczasowych używając operatora działania z przypisaniem:

```
a = a + 20;
b = b * 2;
c = c >> 4;
```

```
a += 20;
b *= 2;
c >>= 4;
```

Tips & tricks

- Korzystaj z listy inicjalizacyjnej konstruktora:

```
MyClass::MyClass (int a, int b, double c)
{
  myA = a;
  myB = b;
  myC = c;
}
```

```
MyClass::MyClass (int a, int b, double c)
  : myA(a)
  , myB(b)
  , myC(c)
{}
```

Tips & tricks

- Uważaj na sposób iteracji po tablicy wielowymiarowej:

```
const unsigned int ROWS = 1000;
const unsigned int COLS = 3000;
int m[ROWS][COLS];

for (int c = 0; c < COLS; ++c)
  for (int r = 0; r < ROWS; ++r)
    m[r][c] = generateValue(r, c);
```

```
const unsigned int ROWS = 1000;
const unsigned int COLS = 3000;
int m[ROWS][COLS];

for (int r = 0; r < ROWS; ++r)
  for (int c = 0; c < COLS; ++r)
    m[r][c] = generateValue(r, c);
```

Tips & tricks

- Korzystaj z instrukcji switch w przypadku rozbudowanych opcji wyboru:

```
if (x == 0) foo();
else if (x == 1) bar();
else if (x == 2) baz();
…
else xyzzy();
```

```
switch (x) {
  case 0: foo(); break;
  case 1: bar(); break;
  case 2: baz(); break;
  …
  default: xyzzy();
}
```

Tips & tricks

- Przenieś pętle do wnętrza funkcji

```
void doSomething ()
{
  //do magic stuff
}

for (int i = 0; i < 1000; ++i)
{
  doSomething();
}
```

```
void doSomething(unsigned int times)
{
  for (int i = 0; i < times; ++i)
    //do magic stuff
}

doSomething(1000);
```

Tips & tricks

- Przekazuj parametry przez referencję
- W miarę możliwości korzystaj z prefiksowej inkrementacji (++i) zamiast postfixowej (i++)
- Wybieraj implementacje algorytmów wykorzystujące iteracje zamiast rekurencji
- Unikaj dynamicznych alokacji w często wywoływanych miejscach
- Używaj obiektów funkcyjnych zamiast wskaźników do funkcji
- Definiuj aliasy dla kontenerów – ewentualna zmiana będzie łatwiejsza
- Unikaj obsługi wyjątków w „gorących miejscach”
- Wyłącz RTTI
- Rozsądnie korzystaj z inline

Strategia zapewnienia wydajności

- Cele wydajnościowe stanowią część specyfikacji
- Dobieraj architekturę i narzędzie zgodnie z wytyczonym celem
- Przemyśl wybór algorytmów i struktur danych przed ich implementacją
- Testy wydajnościowe jako kryterium akceptacyjne
- Automatyzuj testy
- Monitoruj wpływ zmian

Materiały źródłowe

- Valgrind documentation

  http://valgrind.org/

- Perf wiki

  https://perf.wiki.kernel.org/index.php/Main_Page

- Gprof readme

  https://github.com/jrfonseca/gprof2dot/blob/master/README.markdown

- Profiling on Wikipedia

  https://en.wikipedia.org/wiki/Profiling_%28computer_programming%29

Materiały źródłowe

- Optimizing software in C++
  http://www.agner.org/optimize/optimizing_cpp.pdf
- Tips for Optimizing C/C++ Code
  https://people.cs.clemson.edu/~dhouse/courses/405/papers/optimize.pdf
- C++ Optimization Strategies and Techniques
  http://www.tantalon.com/pete/cppopt/main.htm
- Modern Microprocessors: A 90 Minute Guide
  http://www.lighterra.com/papers/modernmicroprocessors/
- GCC documentation
  https://gcc.gnu.org/onlinedocs/