Оптимизация программ с помощью компилятора

(по мотивам книги «Optimizing software in C++» by Agner Fog)

Как компилятор оптимизирует

Далее перечислены некоторые способы оптимизации, применяемые компилятором, которые необходимо знать программисту.

Function inlining – Подстановка функций

Подстановка тела функции вместо ее вызова. Например, следующий код:

```
float square (float a) { return a * a; }
float parabola (float x) { return square(x) + 1.0f; }
```

заменяется таким:

```
float parabola (float x) { return x * x + 1.0f; }
```

Преимущества:

- Устраняются накладные расходы на вызов и возврат из подпрограммы, и на передачу параметров.
- Код кэшируется более эффективно, т.к. становится последовательным.
- Размер кода уменьшается, если вызов функции только один.
- Появляется возможность других оптимизаций.

Недостатки:

- Размер кода увеличивается, если вызовов функции несколько, и функция имеет большой размер.

Компилятор скорее встроит небольшую функцию, или если она вызывается из одного или небольшого числа мест.

Constant folding and constant propagation – Свертка констант и продвижение констант

Выражение или подвыражение, содержащее только константы, заменяется вычисленным результатом. Например, следующий код:

```
double a, b;
a = b + 2.0 / 3.0;
```

заменяется таким:

```
a = b + 0.66666666666666666667;
```

Рекомендуется такие выражения заключать в скобки, чтобы компилятор смог их распознать. Например, `b*(2.0/3.0)` вместо `b*2.0/3.0`.

Константа может быть продвинута через серию выражений. Например, следующий код:

```
float parabola (float x) { return x * x + 1.0f; }
float a, b;
a = parabola (2.0f);
b = a + 1.0f;
```

может быть заменен таким:

```
a = 5.0f;
b = 6.0f;
```

Свертка констант и продвижение констант не могут быть выполнены, если выражение содержи функцию, которая не может быть подставлена или вычислена во время компиляции. Например:

```
double a = sin(0.8);
```

Функция sin определена в отдельной библиотеке, и нельзя ожидать, что компилятор окажется способен ее подставить и вычислить во время компиляции. Некоторые компиляторы способны вычислять такие простые математические функции, как sqrt и pow во время компиляции, но не более сложные, такие как sin.

Pointer elimination – Устранение указателя

Указатель или ссылка могут быть устранены, если компилятору известно, куда ионии указывают. Например, следующий код:

```
void Plus2 (int * p) { *p = *p + 2; }
int a;
Plus2 (&a);
```

может быть заменен таким:

```
a += 2;
```

Common subexpression elimination – Устранение общих подвыражений
Если одно и тоже подвыражение встречается более одного раза, тогда компилятор может вычислить его один раз. Например, следующий код:

```
int a, b, c;
b = (a+1) * (a+1);
c = (a+1) / 4;
```

может быть заменен таким:

```
int a, b, c, temp;
temp = a+1;
b = temp * temp;
c = temp / 4;
```

Register variables – Регистровые переменные
Самые часто используемые переменные хранятся в регистрах. Число доступных целочисленных регистров для 32-битных x86 систем равно 6-7, для 64-битных – 14-15. Число доступных вещественных регистров 32-битных x86 систем равно 8, для 64-битных – 16.
Типичные кандидаты для размещения на регистрах: промежуточные значении выражений, счетчики циклов, параметры функций, указатели, ссылки, указатель ‘this’, значения общих подвыражений и индукционные переменные.
Переменная не может храниться на регистре, если в программе определяется ее адрес, т.е. если есть указатель или ссылка на нее.

Live range analysis – Анализ времени жизни переменной
Определение в коде диапазона, где переменная используется. Оптимизирующий компилятор может использовать один и тот же регистр для нескольких переменных, если их диапазоны использования не пересекаются, или если их значения гарантированно совпадают. Пример:

```
int SomeFunction (int a, int x[]) {
      int b, c;
      x[0] = a;
      b = a + 1;
      x[1] = b;
      c = b + 1;
      return c;
}
```

В этом примере переменные a, b и c могут использовать один и тот же регистр, т.к. их диапазоны использования не пересекаются. Если выражение `c = b + 1` изменить на `c = a + 2`, то переменные a и b не смогут использовать один и тот же регистр, т.к. их диапазоны использования теперь пересекаются.
Компиляторы обычно не используют эту технику для объектов, хранящихся в памяти.

Join identical branches – Объединение одинаковых ветвей
Код может стать более компактным благодаря объединению одинаковых частей кода. Например, следующий код:

```
double x, y, z; bool b;
if (b) {
      y = sin(x);
      z = y + 1.;
}
else {
      y = cos(x);
      z = y + 1.;
}
```

может быть заменен таким:

```
double x, y; bool b;
if (b) { y = sin(x); }
else { y = cos(x); }
z = y + 1.;
```

Eliminate jumps – Устранение переходов

Переходы могут быть устранены путем копирования кода, на который они ведут. Например:

```
int SomeFunction (int a, bool b) {
      if (b) { a = a * 2; }
      else { a = a * 3; }
      return a + 1;
}
```

В данном коде есть переход с `a=a*2` на `return a+1;`. Компилятор может устранить этот переход, копируя оператор return:

```
int SomeFunction (int a, bool b) {
      if (b) {
            a = a * 2;
            return a + 1;
      }
      else {
            a = a * 3;
            return a + 1;
      }
}
```

Переход может быть устранен, если условие является всегда истинным или ложным. Например, следующий код:

```
      if (true) { a = b; }
      else { a = c; }
```

может быть заменен таким:

```
      a = b;
```

Переход также может быть устранен, если значение условного выражения известно из предыдущего перехода. Например, следующий код:

```
      int SomeFunction (int a, bool b) {
            if (b) { a = a * 2; }
            else { a = a * 3; }
            if (b) { return a + 1; }
            else { return a - 1;}
      }
```

может быть заменен таким:

```
      int SomeFunction (int a, bool b) {
            if (b) {
                  a = a * 2;
                  return a + 1;
            }
            else {
                  a = a * 3;
                  return a - 1;
            }
      }
```

Loop unrolling – Раскрутка цикла

…see page 45…

Циклы с малым числом итераций раскрываются полностью. Например, следующий код:

```
      int i, a[2];
      for (i = 0; i < 2; i++) a[i] = i+1;
```

может быть заменен таким:

```
      int a[2];
      a[0] = 1; a[1] = 2;
```

К несчастью, некоторые компиляторы слишком много раскручивают. Излишняя раскрутка циклов не эффективна, т.к. цикл начинает занимать много места в кэше команд и переполняет цикловой буфер, который имеется у некоторых микропроцессоров. В некоторых случаях полезно отключить раскрутку циклов в компиляторе.

Loop invariant code motion – Вынос инвариантного кода за цикл

Вычисление может быть вынесено из цикла, если оно не зависит от счетчика цикла. Например, следующий код:

```
int i, a[100], b;
for (i = 0; i < 100; i++) { a[i] = b * b + 1; }
```
может быть заменен таким:
```
int i, a[100], b, temp;
temp = b * b + 1;
for (i = 0; i < 100; i++) { a[i] = temp; }
```

Induction variables – Индукционные переменные

Выражение, являющееся линейной функцией от счетчика цикла, может быть вычислено путем прибавления константы к предыдущему значению. Например:
```
int i, a[100];
for (i = 0; i < 100; i++) { a[i] = i * 9 + 3; }
```
Компилятор может устранить умножение следующим образом:
```
int i, a[100], temp;
temp = 3;
for (i = 0; i < 100; i++) {
      a[i] = temp;
      temp += 9;
}
```
Индукционные переменные часто используются для вычисления адресов элементов массива. Например:
```
struct S1 { double a; double b; };
S1 list[100]; int i;
for (i = 0; i < 100; i++) {
      list[i].a = 1.0;
      list[i].b = 2.0;
}
```
Для доступа к элементу массива list компилятор должен вычислить его адрес. Адрес элемента list[i] равен адресу начала массива list плюс i*sizeof(S1). Это линейная функция от i, которая может быть вычислена с помощью индукционной переменной. Компилятор может использовать одну и ту же индукционную переменную для доступа к list[i].a и list[i].b. Он также может устранить i и использовать индукционную переменную как счетчик цикла, если финальное значение индукционной переменной вычислить заранее. Получается следующий код:
```
struct S1 { double a; double b; };
S1 list[100], *temp;
for (temp = &list[0]; temp < &list[100]; temp++) {
      temp->a = 1.0;
      temp->b = 2.0;
}
```
Тот факт, что sizeof(S1)=16, скрыт в коде за синтаксисом C++. Целочисленным представлением выражения &list[100] является (int)(&list[100]) = (int)(&list[0]+100*16), и temp++ на самом деле добавляет 16 к целочисленному представлению temp.

Компилятору не необходимости исользовать индукционные перменные для вычисления адресов элементов массива простых типов, потому что процессор имеет аппаратную поддержку для вычисления адреса элемента массива, если адрес может быть представлен как базовый адрес плюс индекс, умноженный на один из множителей: 1, 2, 4 или 8. Если бы a и b в примере были типа float, а не double, тогда sizeof(S1) было бы 8, и индукционной переменной бы не понадобилось, потому что процессор имеет аппаратную поддержку для умножения индекса на 8.

Scheduling – Планирование

Компилятор может переупорядочить команды с целью параллельного исполнения. Например:
```
float a, b, c, d, e, f, x, y;
x = a + b + c;
y = d + e + f;
```
Компилятор может упорядочить вычисление этих двух формул таким образом, что сначала вычислится a+b, затем d+e, затем c добавится к первой сумме, затем f добавится ко второй сумме, затем первй результат запишется в x, и, наконец, второй результат запишется в y. Цель этого – помочь процессору выполнить несколько операций параллельно. Современные процессоры на самом деле могут переупорядочивать инструкции и без помощи компилятора, но компилятор может сделать это переупорядочивание проще для процессора.

Algebraic reductions – Алгебраические преобразования

Большинство компиляторов могут упрощать простые алгебраические выражения, используя фундаментальные законы алгебры. Например, компилятор может заменить -(-a) на a. Такие выражения не часто пишутся программистами, но могут появиться в результате подстановки функций и макроподстановок.

Однако программисты часто пишут выражения, которые могут быть упрощены. Это может быть из-за того, что неупрощенное выражение лучше отражает логику программы, или программист просто не думал о возможности упрощения. Например, программист может предпочитать писать if (!a && !b) вместо if (!(a || b)), даже если у второго выражения на один оператор меньше.

Нельзя ожидать, что компилятор способен упрощать сложные алгебраические выражения. Например, только один компилятор из протестированных смог упростить выражение (a*b*c)+(c*b*a) до a*b*c*2. Довольно трудно реализовать много алгебраических правил в компиляторе. Одни компиляторы могут преобразовывать одни выражения, другие – другие, но ни один не может преобразовывать их все. В случае булевой алгебры возможно реализовать универсальный алгоритм (Quine-McCluskey или Espresso), который может упростить любое выражение, но ни один из протестированных компиляторов так не делает.

Компиляторы лучше упрощают целочисленные выражения, чем вещественные, не смотря на то, что правила алгебры для них одинаковы. Это потому, что алгебраические манипуляции вещественнозначных выражений могут приводить к нежелательным эффектам. Этот эффект может быть продемонстрирован следующим примером:

```
char a = -100, b = 100, c = 100, y;
y = a + b + c;
```

Здесь y получит значение -100+100+100=100. Теперь, в соответствии с правилами алгебры, мы могли бы написать:

```
y = c + b + a;
```

Это могло бы быть полезным, если бы подвыражение c+b могло бы быть использовано еще где-то. В этом примере мы используем 8-битные целые числа с диапазоном от -128 до +127. Целочисленное переполнение заставит значение перейти на другой конец диапазона. Добавление 1 к 127 даст -128, а вычитание 1 из -128 даст 127. Вычисление c+b приведет к переполнению и даст результат -56, а не 200. Далее, мы прибавляем -100 к -56, что опять приведет к переполнению и даст результат 100, а не -156. Неожиданно мы получаем верный результат, т.к. два переполнения на разных концах диапазона нейтрализуют друг друга. По этой причине безопасно использовать алгебраические преобразования над целочисленными выражениями (кроме операторов <, <=, > и >=).

Этот же аргумент не подходит для вещественных выражений. Значения вещественных переменных не переходят на другой конец диапазона при переполнении. Диапазон вещественных значений настолько велик, что нет необходимости заботиться о переполнении, кроме как в специальных математических приложениях. Но необходимо заботиться о потере точности. Повторим пример с вещественными числами:

```
float a = -1.0E8, b = 1.0E8, c = 1.23456, y;
y = a + b + c;
```

Вычисление здесь дает `a+b=0`, и затем `0+1.23456 = 1.23456`. Но мы не получим тот же результат, если мы изменим порядок операндов и сложим b и c. `b+c = 100000001.23456`. Тип float имеет точность примерно семь значащих цифр, так что значение b+c будет округлено до `100000000`. Когда мы прибавим к этому числу a, мы получим 0, а не `1.23456`.

Вывод отсюда таков, что нельзя менять порядок вещественных операндов без опасности потерять точность. Компиляторы этого не делают, если только вы им не укажете специальную опцию, которая допускает менее точные вещественнозначные вычисления. Даже если все остальные опции оптимизации включены, компиляторы не сделают такое, казалось бы, очевидное упрощение как 0/a = 0, так как оно бует неверным, если a равно 0, или бесконечность, или NAN. Различные компиляторы ведут себя по-разному, потому что существуют различные мнения, какие неточности можно допускать, а какие нет.

Нельзя полагаться на то, что компилятор сделает какие-либо арифметические преобразования вещественного кода, и можно рассчитывать только на самые простые преобразования целочисленного кода. Более безопасно делать упрощения вручную.

Devirtualization – Девиртуализация

Оптимизирующий компилятор может пропустить просмотр таблицы виртуальных функций, если он знает, какая версия виртуальной функции необходима. Например:

```
class C0 {
      public:
            virtual void f();
};
class C1 : public C0 {
      public:
            virtual void f();
};
void g() {
      C1 obj1;
      C0 * p = & obj1;
      p->f(); // Virtual call to C1::f
}
```

Без оптимизации компилятору необходимо посмотреть в таблице виртуальных функций, должен ли вызов `p->f();` запустить функцию `C0::f` или `C1::f`. Но оптимизирующий компилятор видит, что p всегда указывает на объект класса C1, так что он может вызвать `C1::f` напрямую, без использования таблицы. К несчастью, немногие компиляторы умеют это делать.

Сравнение нескольких компиляторов

Следующие компиляторы участвовали в сравнительном тестировании:

1. Microsoft C++ Compiler v. 14.00 for 80x86 / x64 (Visual Studio 2005).
2. Borland C++ 5.82 (Embarcadero/CodeGear/Borland C++ Builder 5, 2009).
3. Intel C++ Compiler v. 11.1 for IA-32/Intel64, 2009.
4. Gnu C++ v. 4.1.0, 2006 (Red Hat).
5. PathScale C++ v. 3.1, 2007.
6. PGI C++ v. 7.1-4, 2008.
7. Digital Mars Compiler v. 8.42n, 2004.
8. Open Watcom C/C++ v. 1.4, 2005.
9. Codeplay VectorC v. 2.1.7, 2004.

Никакой разницы в способах оптимизации между 32-битным и 64-битным режимами не было замечено у компиляторов Microsoft, Intel, GNU и PathScale.

Табличка в optimizing_cpp.pdf на страницах 74-76.

Препятствия для оптимизации компилятором

Далее будут рассмотрены некоторые факторы, которые не дают компилятору оптимизировать код требуемым образом.

Не может оптимизировать между модулями

Компилятор не имеет информации о функциях, расположенных в других модулях. Например:

```
module1.cpp
int Func1(int x) {
      return x*x + 1;
}
module2.cpp
int Func2() {
      int a = Func1(2);
      ...
}
```

Если бы Func1 и Func2 были в одном модуле, тогда компилятор смог бы сделать подстановку функции и продвижение константы, и упростил бы переменную a до константы 5. Но у компилятора нет информации о Func1, когда он компилирует module2.cpp.

Самый простой путь решения это проблемы – объединить несколько модулей в один с помощью директивы #include. Это гарантированно будет работать на всех компиляторах. Некоторые компиляторы умеют выполнять оптимизацию между модулями, если указать им специальную опцию компиляции.

Совмещение указателей (pointer aliasing)

При доступе к переменой через указатель или ссылку, компилятор может быть не способен полностью распознать возможность того, что переменная, на которую было указание, идентична некоторой другой переменной в коде. Например:

```
void Func1 (int a[], int * p) {
      int i;
      for (i = 0; i < 100; i++) {
            a[i] = *p + 2;
      }
}
void Func2() {
      int list[100];
      Func1(list, &list[8]);
}
```

Здесь необходимо загрузить *p и вычислить *p+2 сто раз, потому что значение, на которое указывает p, идентично одному из элементов в a[], который будет изменен в цикле. Нельзя считать *p+2 инвариантом цикла, который может быть вынесен за цикл. Конечно, это натянутый пример, но дело в том, что компилятор не может исключить существование таких примеров. Поэтому компилятор не имеет права считать *p+2 инвариантом цикла и вынести его за цикл.

Большинство компиляторов имеют опцию, включающую предположение, что совмещения указателей нет (/Oa). Самый простой способ преодолеть эту проблему – включить эту опцию. Это означает, что вы тщательно изучили все указатели и ссылки в коде чтобы убедиться, что ни к одному объекту не существует более одного способа доступа в одном месте кода. Также возможно сказать компилятору, что конкретный указатель ни с чем не пересекается, с помощью ключевого слова __restrict или __restrict__, если это поддерживается компилятором.

Мы никогда не можем быть уверены, что компилятор принял подсказку о несовмещении указателей. Единственный способ убедиться, что код оптимизирован, это сделать это самому. В примере выше можно было запомнить значение *p+2 во временной переменной до цикла, если вы уверены, что указатель не ссылается на какой-либо элемент массива. Этот метод требует того, чтобы вы могли предсказать, где могут быть препятствия для оптимизации.

Динамическое выделение памяти

Если массив или объект выделен динамически, доступ к нему осуществляется через указатель. Для программиста может быть очевидно, что указатели на различные динамически выделенные объекты не пересекаются, но компилятор обычно этого не видит. Динамическое выделение также не позволяет компилятору оптимально выровнять данные. Кроме того, компилятор не знает, выровнены ли данные. Предпочтительно объявлять объекты и массивы фиксированного размера внутри функций, в которых они нужны.

Чистые функции (pure functions)

Чистая функция – это функция, которая не имеет побочных эффектов, и ее возвращаемое значение зависит только от значений аргументов. Это близко соответствует математическому понятию «функция».

Множественные вызовы чистой функции с одними и теми же аргументами гарантированно дадут один и тот же результат. Компилятор может вынести общие подвыражения, которые содержат вызов чистой функции, и он может вынести за цикл инвариантный код, содержащий вызов чистой функции. К несчастью, компилятор не может знать, что функция является чистой, если она объявлена в другом модуле или библиотеке. В таких случаях приходится делать вручную такие оптимизации, как вынос общих подвыражений, продвижение констант и вынос инвариантного кода за цикл.

У компилятора GNU и компилятора Intel под Linux есть атрибут, который можно использовать с прототипом функции для указания того, что эта функция чистая. Например:

```
#ifdef __GNUC__
#define pure_function __attribute__((const))
#else
#define pure_function
#endif
double Func1(double) pure_function ;
double Func2(double x) { return Func1(x) * Func1(x) + 1.; }
```

Здесь компилятор GNU сделает только один вызов Func1, тогда как другие компиляторы сделают два. Некоторые другие компиляторы (Microsoft, Intel) знают, что стандартные библиотечные функции, такие как sqrt, pow, log являются чистыми, однако нет способа сказать этим компиляторам, что определенная пользователем функция – чистая.

Виртуальные функции и указатели на функции
Это редкость для компилятора точно предсказать, какая версия виртуальной функции будет вызвана, или на какую функцию указывает указатель. Таким образом, он не может сделать подстановку функции или иным способом оптимизировать через вызов функции.

Алгебраические преобразования
Большинство компиляторов могут делать простые алгебраические преобразования, такие как -(-a) = a, но они не могут делать более сложные преобразования. Алгебраическое преобразование – это сложный процесс, который трудно реализовать в компиляторе.
Многие алгебраические преобразования недопустимы по причине математической чистоты. Во многих случаях возможно сконструировать странные примеры, в которых преобразование приведет к переполнению или потери точности, особенно в вещественных выражениях. Компилятор не может выявить возможность того, что конкретное преобразование будет неверным в конкретной ситуации, но программист может. Таким образом, во многих случаях необходимо делать алгебраические преобразования явно.
Целочисленные выражения менее восприимчивы к проблемам переполнения и потери точности. Поэтому компилятор имеет больше возможностей по преобразованию целочисленных выражений, чем вещественных. Большинство преобразований, включающих сложение, вычитание и умножение, допустимы во всех случаях, тогда как многие преобразования, включающие деление и операторы сравнения (например, '>') не допустимы по причине математической чистоты. Например, компилятор не может упростить выражение -a > -b до a < b из-за очень неопределенной возможности переполнения.

Вещественные индукционные переменные
Компиляторы не могут создавать вещественные индукционные переменные по той же причине, по которой они не могут выполнять алгебраические преобразования вещественных выражений. Тем не менее, это необходимо делать вручную. Этот принцип полезен если функцию от счетчика цикла проще вычислить из прошлого значения, чем из счетчика цикла. Любое выражение, являющееся полиномом n-й степени счетчика цикла может быть вычислено с помощью n сложений и без умножений. Следующий пример показывает принцип для полинома 2-й степени:

```
// Loop to make table of polynomial
const double A = 1.1, B = 2.2, C = 3.3; // Polynomial coefficients
double Table[100]; // Table
int x; // Loop counter
for (x = 0; x < 100; x++) {
      Table[x] = A*x*x + B*x + C; // Calculate polynomial
}
```

Вычисление этого полинома может быть вычислено всего с двумя сложениями, используя две индукционные переменные:

```
// Calculate polynomial with induction variables
const double A = 1.1, B = 2.2, C = 3.3; // Polynomial coefficients
double Table[100]; // Table
int x; // Loop counter
const double A2 = A + A; // = 2*A
double Y = C; // = A*x*x + B*x + C
double Z = A + B; // = Delta Y
for (x = 0; x < 100; x++) {
      Table[x] = Y; // Store result
      Y += Z; // Update induction variable Y
      Z += A2; // Update induction variable Z
}
```

Цикл в примере имеет две цепочки зависимостей, а именно две индукционные переменные Y и Z. Каждая цепочка зависимостей имеет такую же латентность, как и латентность вещественной операции сложения. Это достаточно мало, чтобы оправдать метод. Более длинная цепочка зависимостей может

сделать метод индукционных переменных неподходящим, разве что новое значение будет вычисляться из позапрошлого или более раннего значения.
Метод индукционных переменных также может быть векторизован, если принять во внимание, что каждое значение вычисляется из значения, лежащего на r позиций раньше в последовательности, где r – это число элементов в векторе или степень раскрутки цикла. Немного математических выкладок требуется, чтобы в каждом случае найти правильную формулу.

Встроенная функция имеет невстроенную копию
Встроенная функция может быть вызвана из другого модуля. Компилятор вынужден сделать невстроенную копию встроенной функции. Эта невстроенная функция является мертвым кодом, если она никем не вызывается. Такие функции повышают фрагментацию кода, в результате чего кэш команд работает менее эффективно.
Есть разные способы решения этой проблемы. Если функция не вызывается из других модулей, добавьте к ее определению ключевое слово static. Это укажет компилятору, что функция не может быть вызвана из другого модуля. Определение функции как static упрощает для компилятора определение того, является ли оптимальным встраивание этой функции, и предотвращает создание невстроенной копии. Ключевое слово static также делает возможными разные другие оптимизации, потому что компилятор не должен следовать каким-либо специальным соглашениям на вызов функций. Его можно добавлять ко всем локальным функциям (не методам класса).
К сожалению, этот метод не работает для методов классов, потому что для них слово static имеет другое значение. Вы можете заставить метод быть встроенным, описав его тело внутри определения класса. Это не даст компилятору создать невстроенную копию, но недостаток в том, что метод встраивается всегда, даже если это не оптимально.
Некоторые компиляторы имеют специальную опцию, которая позволяет линковщику удалить все функции, к которым нет обращений. Рекомендуется ее включить.

Препятствия для оптимизации процессором
Современные процессоры могут делать много оптимизаций, исполняя команды вне порядка. Длинные цепочки зависимостей в коде не позволяют процессору исполнять команды вне порядка. Нужно их избегать, особенно в циклах.

Опции оптимизации компилятора