Новосибирский государственный университет
Факультет информационных технологий
Кафедра параллельных вычислений

Эффективное программирование современных микропроцессоров и мультипроцессоров

Возможности компиляторов по оптимизации программ

Преподаватели:
Киреев С.Е.
Калгин К.В.

ТИПЫ ОПТИМИЗАЦИЙ КОМПИЛЯТОРА ПО СФЕРЕ ПРИМЕНЕНИЯ

Типы оптимизаций компилятора по сфере применения

- Peelhole-оптимизация
  - Рассматривается несколько соседних инструкций
- Локальная оптимизация
  - Рассматривается один базовый блок
- Внутрипроцедурная оптимизация
  - Рассматривается одна подпрограмма
- Оптимизация циклов
  - Рассматривается один или несколько вложенных циклов
- Межпроцедурная оптимизация
  - Рассматривается весь исходный код программы

ТЕХНИКИ ОПТИМИЗАЦИИ, ПРИМЕНЯЕМЫЕ КОМПИЛЯТОРАМИ

Подстановка функций
Function inlining

- Подстановка тела функции вместо вызова
  - Применяется для небольших функций, вызываемых из небольшого числа мест.
- Пример:

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

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

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

Подстановка функций
Function inlining

- 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.

- Способы устранения невстроенной копии функции:
  - Объявить функцию как static
    - Делает возможными другие оптимизации
  - Некоторые компиляторы имеют специальную опцию, которая позволяет линковщику удалить все функции, к которым нет обращений.

Свертка и распространение констант
Constant folding and constant propagation

- Выражение, содержащее только константы, заменяется вычисленным результатом

  a = b + 2.0/3.0; → a = b + 0.666666666666667;

  – Рекомендуется такие выражения заключать в скобки

    b*2.0/3.0 → (b*2.0)/3.0

    b*(2.0/3.0) → b*0.666666666666667

  – Константа может быть продвинута через серию выражений

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

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

  – Свертка и распространение констант не могут быть выполнены, если выражение содержит функцию, которая не может быть подставлена или вычислена во время компиляции.

Устранение указателя
Pointer elimination

- Указатель или ссылка могут быть устранены, если компилятору известно, куда они указывают.

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

```
a += 2;
```

Устранение общих подвыражений
Common subexpression elimination

- Если одно и то же подвыражение встречается более одного раза, тогда компилятор может вычислить его один раз.

```
int a, b, c;

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

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

Регистровые переменные
Register variables

- Самые часто используемые переменные компилятор размещает на регистрах (для всей области жизни переменной или для её части)

- Типичные кандидаты на размещение на регистрах:
  - Промежуточные значения в выражениях, счётчики циклов, параметры функций, указатели, ссылки, указатель this, общие подвыражения, индукционные переменные.

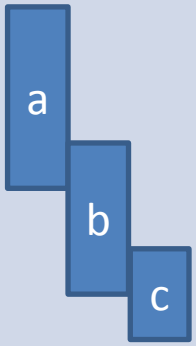
- Переменная размещается в памяти, если
  - Определяется её адрес (есть указатель или ссылка на неё)
  - Она объявлена как глобальная или статическая

- Переменная не может быть временно размещена на регистре, если
  - Она была объявлена как volatile

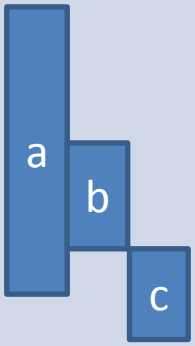
Анализ времени жизни переменной
Live range analysis

- Оптимизирующий компилятор может использовать один и тот же регистр для нескольких переменных, если их диапазоны использования не пересекаются, или если их значения гарантированно совпадают.

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




```
int SomeFunction (int a, int x[])
{
  int b, c;
  x[0] = a;
  b = a + 1;
  x[1] = b;
  c = a + 2;
  return c;
}
```



- Для объектов в памяти эта техника не используется – для каждого объекта выделяется своя область памяти.

Объединение одинаковых ветвей
Join identical branches

- Код может стать более компактным благодаря объединению одинаковых частей кода.

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

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

Устранение переходов
Eliminate jumps

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

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

```
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.

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

```
a = b;
```

Устранение переходов
Eliminate jumps

- Переход также может быть устранен, если значение условного выражения известно из предыдущего перехода.

```
int SomeFunction (int a, bool b) {

  if (b) { a = a * 2; }
  else   { a = a * 3; }

  if (b) { return a + 1; }
  else   { return a - 1;}
}
```

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

Раскрутка цикла
Loop unrolling

- Уменьшение числа итераций с увеличением тела цикла
  - Плюсы
    - Уменьшаются накладные расходы на организацию цикла,
    - Появляются возможности для дополнительных оптимизаций
  - Минусы
    - Тело цикла может стать слишком большим, занять слишком много места в кэше команд и не поместиться в цикловой буфер.

- Циклы с малым числом итераций могут быть раскрыты полностью.

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

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

Вынос инвариантного кода за цикл
Loop invariant code motion

- Вычисление может быть вынесено из цикла, если оно не зависит от счетчика цикла.

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

for (i = 0; i < 100; i++) {
  a[i] = b * b + 1;
}
```

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

Индуктивные переменные
Induction variables

- Выражение, являющееся линейной функцией от счетчика цикла, может быть вычислено путем прибавления константы к предыдущему значению.

```
int i, a[100];
for (i = 0; i < 100; i++) {
  a[i] = i * 9 + 3;
}
```

```
int i, a[100], temp;
temp = 3;
for (i = 0; i < 100; i++) {
  a[i] = temp;
  temp += 9;
}
```

- Индуктивные переменные часто используются для вычисления адресов элементов массива.

```
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;
}
```

```
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;
}
```

Планирование
Scheduling

- Компилятор может переупорядочить команды с целью параллельного исполнения.

```
float a, b, c, d, e, f, x, y;
x = a + b + c;
y = d + e + f;
```

```
float a, b, c, d, e, f, x, y;
ab = a + b;
de = d + e;
x = ab + c;
y = de + f;
```

Алгебраические преобразования
Algebraic reductions

- Большинство компиляторов могут упрощать простые алгебраические выражения, используя фундаментальные законы алгебры.
- Компиляторы лучше упрощают целочисленные выражения, чем вещественные из-за опасности потери точности.

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

-100 100 → 0; 0 100 → 100

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

100 100 → 200; 200 → -56; -56 -100 → -156 → 100

переполнение

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

-1.0E8 1.0E8 → 0.0; 0.0 1.2345 → 1.2345

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

1.0E8 1.2345 → 100000001.23456 → 1.0E8; 1.0E8 -1.0E8 → 0

потеря точности

Алгебраические преобразования
Algebraic reductions

- Компиляторы меняют порядок вещественных операндов только при указании специальной опции
  - gcc -ffast-math
  - icc -fp-model fast[=1|2]
- Некоторые «очевидные» преобразования вещественных выражений компиляторы не делают:
  - Не создаёт индуктивных переменных
  - 0.0 / a = 0.0
  - …

- 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.

Девиртуализация
Devirtualization

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

```
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
}
```

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.
- Некоторые компиляторы знают, что стандартные функции (sqrt, pow, log, …) являются чистыми. Компиляторы gcc и icc позволяют объявить чистыми пользовательские функции:

```
__attribute__((pure)) double Func1(double);

__attribute__((const)) double Func2(double);

double Func3(double x) {
  return Func1(x) * Func1(x) + Func2(x) * Func2(x);
}
```

ЗАВИСИМОСТИ ПО ДАННЫМ (ПО РЕСУРСАМ)

Зависимости по данным

- Зависимости препятствуют оптимизации:
  - Зависимые команды нельзя переупорядочить или выполнить одновременно (при векторизации и распараллеливании компилятором, при выполнении в суперскалярном процессоре)

Зависимости по данным

- Виды зависимостей по данным (по ресурсам)
  - Flow (True) dependence – Read After Write
    - X := …
    - … := f(X)
  - Anti dependence – Write After Read
    - … := f(X)
    - X := …
  - Output dependence – Write After Write
    - X := …
    - X := …
  - Input dependence – Read After Read
    - … := f(X)
    - … := g(X)

Зависимости по данным

- Виды зависимостей по данным (по ресурсам)
  - Flow (True) dependence – Read After Write
    - X := …
    - … := f(X)

    истинная зависимость – нельзя устранить
  - Anti dependence – Write After Read
    - … := f(X)
    - X := …

    можно устранить
  - Output dependence – Write After Write
    - X := …
    - X := …

    можно устранить
  - Input dependence – Read After Read
    - … := f(X)
    - … := g(X)

    не нужно устранять

Зависимости по данным

- Виды зависимостей в цикле
  - Loop-independent – зависимость внутри одной итерации цикла

```
for (i=0; i<N; i++) {
  a[i] = b[i] + 1;
  c[i] = a[i] * b[i];
}
```

  - Loop-carried – зависимость между различными итерациями цикла

```
for (i=0; i<N; i++) {
  a[i] = b[i] + 1;
  c[i] = a[i] + c[i+1];
}
```

Зависимости по данным

- Способы устранения Anti и Output dependences
  - Variable Renaming
  - Scalar Expansion
  - Node Splitting

Зависимости по данным

- Устранение Anti и Output dependences: **Variable Renaming**

A = X + B
X = Y + 1
C = X + B
X = Z + B
D = X + 1

Зависимость между всеми операциями,
так как используется общая переменная.

Зависимости по данным

- Устранение Anti и Output dependences: **Variable Renaming**

```
A = X + B
X = Y + 1
C = X + B
X = Z + B
D = X + 1
```

```
A = X + B
X1 = Y + 1
C = X1 + B
X2 = Z + B
D = X2 + 1
```

Независимые группы операций

- Компилятор и процессор это распознают и применяют.

Зависимости по данным

- Устранение Anti и Output dependences: **Scalar Expansion**

```
for (i=0; i<N; i++) {
  x = a[i] + 1;
  b[i] = x * x;
}
```

Зависимость между всеми итерациями,
так как используется общая переменная.

Зависимости по данным

- Устранение Anti и Output dependences: **Scalar Expansion**

```
for (i=0; i<N; i++) {
  x = a[i] + 1;
  b[i] = x * x;
}
```

```
for (i=0; i<N; i++) {
  x[i] = a[i] + 1;
  b[i] = x[i] * x[i];
}
```

Итерации независимы, так как своя переменная на каждой итерации.

Зависимости по данным

- Устранение Anti и Output dependences: **Scalar Expansion**

```
for (i=0; i<N; i++) {
  x = a[i] + 1;
  b[i] = x * x;
}
```

```
for (i=0; i<N; i++) {
  x[i] = a[i] + 1;
  b[i] = x[i] * x[i];
}
```

```
for (i=0; i<N; i++) {
  float x = a[i] + 1;
  b[i] = x * x;
}
```

- Компилятор это распознаёт и применяет.

Зависимости по данным

- Устранение Anti и Output dependences: **Node Splitting**

```
for (i=0; i<N; i++) {
  a[i] = b[i] + 1;
  c[i] = (a[i] + a[i+1])/2;
}
```

Последовательность операций фиксирована, так как последующие операции затирают входные данные предыдущих.

Зависимости по данным

- Устранение Anti и Output dependences: **Node Splitting**

```
for (i=0; i<N; i++) {
  a[i] = b[i] + 1;
  c[i] = (a[i] + a[i+1])/2;
}
```

```
for (i=0; i<N; i++) {
  temp[i] = a[i+1];
  a[i] = b[i] + 1;
  c[i] = (a[i] + temp[i])/2;
}
```

Можно выполнять каждую операцию сразу для нескольких итераций (например, при векторизации)

Зависимости по данным

- Устранение Anti и Output dependences: **Node Splitting**

```
for (i=0; i<N; i++) {
  a[i] = b[i] + 1;
  c[i] = (a[i] + a[i+1])/2;
}
```

```
for (i=0; i<N; i++) {
  temp[i] = a[i+1];
  a[i] = b[i] + 1;
  c[i] = (a[i] + temp[i])/2;
}
```

```
for (i=0; i<N; i++) {
  float temp = a[i+1];
  a[i] = b[i] + 1;
  c[i] = (a[i] + temp)/2;
}
```

СРАВНЕНИЕ КОМПИЛЯТОРОВ

Сравнение компиляторов

- Agner Fog «Optimizing software in C++», p.74
- Список компиляторов:
  - Microsoft C++ Compiler v. 14.00 for 80x86 / x64 (Visual Studio 2005).
  - Borland C++ 5.82 (Embarcadero/CodeGear/Borland C++ Builder 5, 2009).
  - Intel C++ Compiler v. 11.1 for IA-32/Intel64, 2009.
  - Gnu C++ v. 4.1.0, 2006 (Red Hat).
  - PathScale C++ v. 3.1, 2007.
  - PGI C++ v. 7.1-4, 2008.
  - Digital Mars Compiler v. 8.42n, 2004.
  - Open Watcom C/C++ v. 1.4, 2005.
  - Codeplay VectorC v. 2.1.7, 2004.
- Вывод: нет компилятора, который бы делал все оптимизации.

ПРЕПЯТСТВИЯ ДЛЯ ОПТИМИЗАЦИИ КОМПИЛЯТОРОМ

Невозможность оптимизировать между модулями

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

module1.cpp

```
int Func1(int x) {
  return x*x + 1;
}
```

module2.cpp

```
int Func2() {
  int a = Func1(2);
  ...
}
```

- Некоторые компиляторы умеют выполнять оптимизацию между модулями при указании специального ключа
  - icc -ipo

Перекрытие 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.

```
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]);
}
```

Компилятор не имеет права считать *p+2 инвариантом цикла

- Как сказать компилятору, что пересечения указателей нет:
  - Специальная опция:
    - gcc -fstrict-aliasing - включается при -O2, -O3, -Os
    - icc -fno-alias / -fansi-alias
  - Ключевое слово __restrict / __restrict__ (если компилятор поддерживает)

Виртуальные функции и указатели на функции

- Компилятор редко может 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.

ПРЕПЯТСТВИЯ ДЛЯ ОПТИМИЗАЦИИ ПРОЦЕССОРОМ

Длинные цепочки зависимостей

- Современные процессоры могут выполнять инструкции параллельно вне порядка. Длинные цепочки зависимых команд не позволяют это делать. Нужно их избегать, особенно с долгими операциями, особенно в циклах.

Литература

- Agner Fog, Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms http://www.agner.org/optimize/optimizing_cpp.pdf