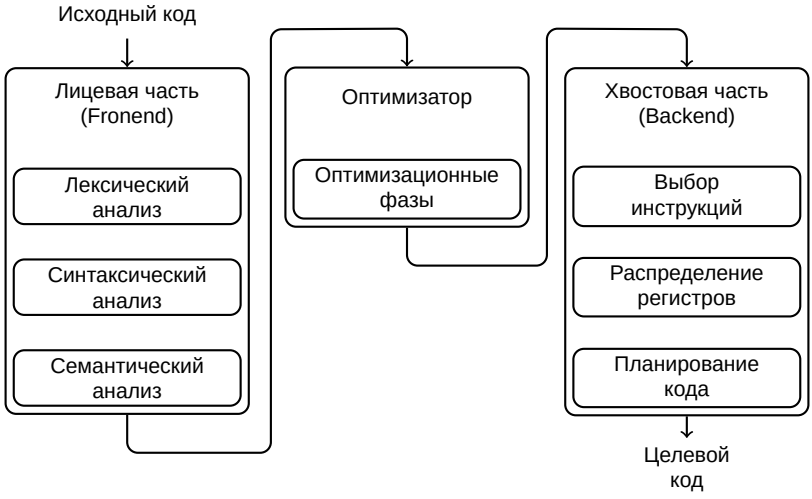
МЦСТ

Устройство современных компиляторов

Лекция 2.
Оптимизации компилятора.

Маркин А. Л.
alexanius@gmail.com

Схема работы компилятора



Часть 1.
Генерация кода.

Генерация кода

Генерация кода — процесс перевода программы из промежуточного представления в целевой код.

Обычно целевым кодом является ассемблер машины, на которой этот код будет исполняться.

Иногда, особенно для динамических компиляторов необходимо уметь создавать код сразу в виде исполняемых процессором инструкций.

Генерация кода

В низкоуровневом представлении уже известно расположение объектов в памяти. Часть из них переносится на регистры, часть на стек (объекты лежащие на куче, просто содержат свой адрес в регистрах).

Представление на ранних фазах:

```
t1 <- d - a
t2 <- c[i]
t3 <- b * t3
e  <- t1  + t2
```

Представление перед кодогенерацией:

```
Vs6 <- Vs3 - Vs0
Vs7 <- Vs2[Vs5]
Vs8 <- Vs1 * Vs7
Vs4 <- Vs6 + Vs7
```

На текущем этапе компилятор оперирует «виртуальными» регистрами, которых может быть бесконечно много.

Выбор инструкций

Выбор инструкций — первая стадия работы генератора кода. На ней мы заменяем операции промежуточного представления реальными операциями. При этом аргументы операций всё ещё принадлежат представлению.

Представление перед кодогенерацией:

```
Vs6 <- Vs3 - Vs0
Vs7 <- Vs2[Vs5]
Vs8 <- Vs1 * Vs7
Vs4 <- Vs6 + Vs7
```

Представление после выбора инструкций:

```
fsubs Vs3, Vs0, Vs6
lds   Vs2, Vs5, Vs7
fmuls Vs1, Vs7, Vs8
fadds Vs6, Vs7, Vs4
```

Планирование

Планирование — процесс переупорядочивания инструкций для определения последовательности их исполнения. Этот процесс необходим для того чтобы процессор мог задействовать максимальное количество исполняющих устройств.

Представление перед планированием:

```
0: fsubs Vs3, Vs0, Vs6
1: lds   Vs2, Vs5, Vs7
4: fmuls Vs1, Vs7, Vs8
8: fadds Vs6, Vs7, Vs4
```

Представление после планирования:

```
0: lds   Vs2, Vs5, Vs7
0: fsubs Vs3, Vs0, Vs6
3: fmuls Vs1, Vs7, Vs8
7: fadds Vs6, Vs7, Vs4
```

Планирование

Планирование инструкций осуществляется только в пределах одного линейного участка графа потока управления (хотя бывает и глобальное планирование).

Чем больше линейный участок, тем больше возможностей для планирования и реализации параллелизма уровня инструкций.

Любой вызов функции или ветвление завершают линейный участок. Чем больше ветвлений и вызовов в горячем коде, тем хуже его оптимизирует компилятор и тем медленнее он исполняется.

Распределение регистров

Распределение регистров — отображение множества виртуальных регистров на множество физических регистров микропроцессора.

Представление перед распределением регистров:

```
0: lds   Vs2, Vs5, Vs7
0: fsubs Vs3, Vs0, Vs6
3: fmuls Vs1, Vs7, Vs8
7: fadds Vs6, Vs7, Vs4
```

Представление после распределения регистров:

```
0: lds   r2, r5, r7
0: fsubs r3, r0, r6
3: fmuls r1, r7, r8
7: fadds r6, r7, r4
```

Распределение регистров

Для регистра существует понятие **времени жизни** — время от момента определения регистра до конца его использования. В течении этого времени на данный физический регистр нельзя распределять другие виртуальные регистры.

Максимальная степень перекрытия в такте — максимальное количество физических регистров, которые потребуются для распределения.

Например, в процессорах Эльбрус количество таких регистров составляет 192.

Распределение регистров

Часто бывает ситуация когда максимальная степень перекрытия регистров больше чем доступных физических регистров в процессоре.

В этом случае для регистров, не используемых в данный момент приходится делать операцию сброса в память и потом поднятия из памяти (spill и fill).

Эта операция значительно замедляет исполнение, однако без неё работа программы невозможна.

Часть 2.
Секция 1. Оптимизации компилятора.

Оптимизирующие компиляторы

В 80-х годах стало выделяться два вида компиляторов:

1. **Неоптимизирующие (отладочные)** — нацелены на максимально быструю компиляцию и простую отладку. Почти не перемешивают код, что позволяет быстро находить ошибки в исходных программах.
2. **Оптимизирующие** — ценой увеличения времени компиляции и крайне затруднительной отладки уменьшают время исполнения программы и используют возможности целевой платформы.

Определение оптимизаций

Оптимизация — преобразование программы, нацеленное на улучшение её характеристик.

При проведении оптимизации необходимо учитывать несколько факторов:

- **Безопасность** — оптимизация не должна менять поведение программы на любых входных данных.
- **Целесообразность** — оптимизация должна приносить определённую выгоду (ускорение программы, возможность применения других оптимизаций), при этом выгода должна быть адекватна времени, потраченному на компиляцию.

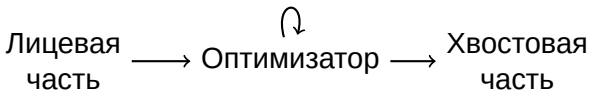
Определение оптимизаций

Что можно добиться при помощи оптимизаций:

- Увеличение производительности
- Уменьшение размера исполняемого файла
- Уменьшение времени компиляции
- Обнаружение ошибок

Определение оптимизаций

Во время оптимизации компилятор последовательно применяет к программе каждую оптимизацию из заданного набора и потом передаёт получившийся результат на кодогенерацию.



За время компиляции оптимизатор может сделать более 400 оптимизирующих проходов по представлению, некоторые из которых сами по себе являются наборами проходов.

Определение оптимизаций

Каждый проход оптимизатора по представлению называется **фазой** (pass).

Фаза не обязательно подразумевает оптимизирующее преобразование. Она может содержать просто **анализ** представления, выявляющий свойства программы с целью дальнейшего их использования в оптимизирующих преобразованиях.

Некоторые преобразования программы не являются оптимизирующими и носят чисто технический характер.

Определение оптимизаций

Важно понимать что оптимизации редко бывают универсальными. Одна и та же оптимизация может ускорять одни примеры кода и замедлять другие.

Оптимизатор не может гарантировать что применение конкретной оптимизации не замедлит исполняемый код на практике!

Часть 2.
Секция 2. Классификация оптимизаций.

Виды оптимизаций

Оптимизации могут классифицироваться по зависимости от аппаратной платформы:

- **Архитектурно-независимые** — универсальные оптимизации, не опирающиеся на возможности целевой платформы.
- **Архитектурно-зависимые** — оптимизации, использующие возможности целевой платформы для ускорения итоговой программы.

Виды оптимизаций

Оптимизации применяются к различным областям программы:

- **Локальные** — оптимизации применяются внутри одного линейного участка. Обычно это простые оптимизации, нацеленные на устранение избыточности кода и использование аппаратных возможностей платформы.
- **Регионные** — оптимизации применяются к набору линейных участков внутри графа потока управления. Такие оптимизации могут быть направлены на ускорение работы циклов, использование аппаратных возможностей, устранение избыточности, применение архитектурно-независимых оптимизаций.
- **Глобальные (внутрипроцедурные)** — оптимизации применяются ко всей процедуре, могут оценивать её характеристики и принимать решение об эффективности тех или иных оптимизаций.

Виды оптимизаций

Оптимизации применяются к различным областям программы:

- **Межпроцедурные** — оптимизации используют граф вызовов и оптимизируют взаимодействие процедур друг с другом. Такие оптимизации являются архитектурно-независимыми.
- **Межмодульные** — расширение области видимости межпроцедурных оптимизаций на всю программу. Для их применения необходимо применять режим **вся программа**, объединяющий все модули компилируемой программы в один.

Часть 3.
Секция 1. Локальные оптимизации.

Примеры локальных оптимизаций

Локальные оптимизации действуют в пределах линейного участка и нацелены на устранение избыточности вычислений и на вычисления во время компиляции.

Примеры таких оптимизаций:

- Оконные оптимизации (peephole) — набор оптимизаций, включающий в себя различные упрощения выражений и простые вычисления с константами.
- Сбор общих подвыражений — оптимизации, объединяющие повторяющиеся вычисления
- Удаление мёртвого кода
- Переупорядование и балансировка выражений — оптимизации, помогающие другим оптимизациям и наращивающие параллелизм уровня инструкций.

Примеры локальных оптимизаций

Работа **свёртки констант**:

```
3 + 2 = 5
i + 0 = 0 + i = i - 0 = i
0 - i = -i
i * 1 = 1 * i = i / 1 = i
i * 0 = 0 * i = 0
-(-i) = i
i + (-j) = i - j
b or true = true or b = true
b or false = false or b = b
b and true = true and b = b
b and false = false and b = false
```

Примеры локальных оптимизаций

Работа **алгебраических упрощений**:

```
pow(x, 2)  →          x * x
  x * 2    →          x + x
  x * 5    →  t = x << 2; t += x
  x * 7    →  t = x << 3; t -= x
```

Примеры локальных оптимизаций

Работа **распространения констант**:

```
x <- 10          x <- 10
y <- x + 5   →   y <- 10 + 5
```

В данном примере после распространения констант следует запускать свёртку констант. Бывают обратные ситуации или (чаще) ситуации когда следует поочерёдно запускать эти оптимизации пока избыточность кода не будет устранена.

Примеры локальных оптимизаций

Некоторые упрощения могут влиять на результат работы программы:

```
x = y * (1 / z)  →  x = y * z
```

Примеры локальных оптимизаций

Некоторые упрощения могут влиять на результат работы программы:

```
x = y * (1 / z)  →  x = y * z
```

Данное преобразование корректно математически, но результат в случае вычислений с плавающей точкой будет различаться

Примеры локальных оптимизаций

Когда бывает полезно устранять общие подвыражения:

```
f(int * p, int * q, int ind)
   for(int i = i; i < N; i++)
      p[ind * 4 + i] = q[ind * 4 + i];
```

Представление до и после оптимизации:

```
t0    <- ind * 4
t1    <- t1  + i
t2    <- q[t1]
t3    <- ind * 4
t4    <- t3  + i
p[t4] <- t2
```

$\rightarrow$

```
t0    <- ind * 4
t1    <- t1  + i
t2    <- q[t1]
p[t1] <- t2
```

Часть 3.
Секция 2. Оптимизации потока управления.

Примеры оптимизаций потока управления

Для работы локальных оптимизаций важно иметь большие линейные участки, в рамках которых они работают.

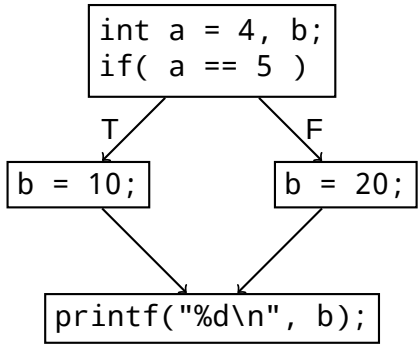
Реальный код обычно состоит из большого количества ветвлений, которые создают избыточные переходы и мешают работать локальным оптимизациям и планированию.

Обычно оптимизации **потока управления** стремятся уменьшить накладные расходы, связанные с передачей управления и увеличить возможности компилятора по оптимальному планированию инструкций.

Примеры оптимизаций потока управления

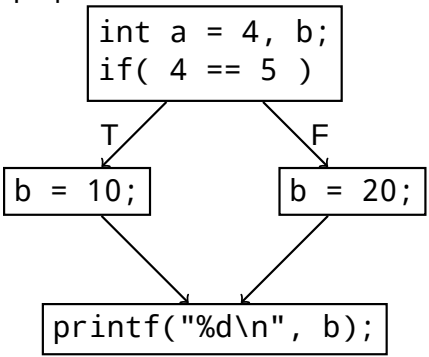
Представим следующий код:

```
int a = 4, b;

if( a == 5 )
   b = 10;
else
   b = 20;

printf("%d\n", b);
```

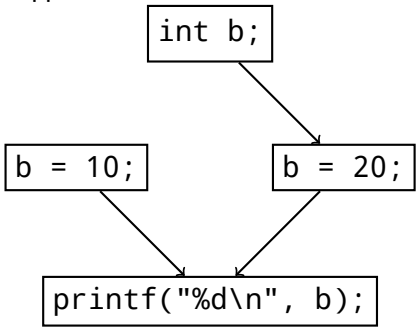


Примеры оптимизаций потока управления

После распространения констант получаем следующий граф:



После удаления мёртвого кода:



Примеры оптимизаций потока управления

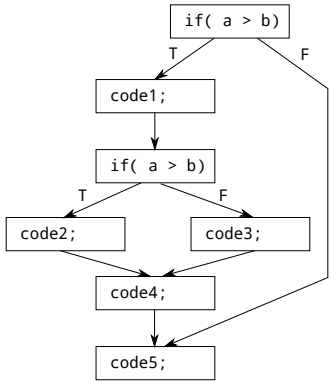
После удаления недостижимого кода, устранения безусловных переходов и распространения констант:

```
printf("%d\n", 20);
```

Можно видеть что сначала локальные оптимизации создали условие для применения оптимизаций управления, которые в свою очередь создали условия для применения локальных оптимизаций.

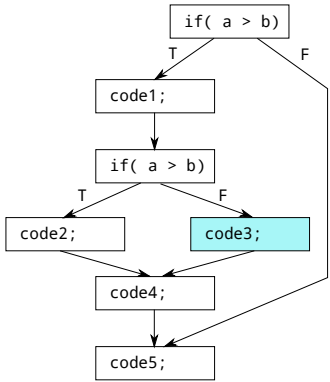
Примеры оптимизаций потока управления

Иногда в ходе оптимизаций граф потока управления приходит к следующему виду:

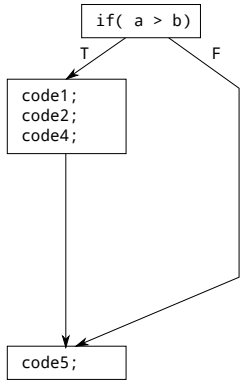


Примеры оптимизаций потока управления

Блок кода с `code3;` никогда не не исполнится



После упрощений:



Часть 3.
Секция 3. Оптимизация работы с памятью.

Примеры оптимизация работы с памятью

Часто основная нагрузка приложения ложится на работу с памятью. Память работает значительно медленнее процессора, поэтому исполняющим устройствам приходится ждать подгрузки данных.

Задачей оптимизации работы с памятью является минимизация времени ожидания данных, и, как следствие, ускорение работы приложения.

Примеры оптимизация работы с памятью

Расположение объектов в памяти:

```
void foo(void) {
    int i; // Регистр
    int j; // Стек
    struct {
        int * a;
        float * b; } str; // Стек

    str.a = &j;
    str.b = malloc(sizeof(float) * 100); // Куча
    // ...
    free(str.b);
}
```

Примеры оптимизация работы с памятью

Обычно объекты, которые можно разместить на регистрах, компилятор стремится разместить на регистрах (с учётом ограничения их количества и целесообразности).

Если на объект брался указатель, то его нельзя перекладывать на регистр — он всегда должен иметь адрес в памяти.

Существует набор оптимизаций, стремящийся преобразовать расположение объектов в памяти с целью ускорения доступа к данным.

Примеры оптимизация работы с памятью

Простейший пример — разрезание структуры:

```
void foo(void) {
    int     i       // Регистр
    int     j       // Стек
    int   * .str.a; // Регистр
    float * .str.b; // Регистр

    .str.a = &j;
    .str.b = malloc(sizeof(float) * 100); // Куча
    // ...
    free(.str.b);
}
```

Примеры оптимизация работы с памятью

Вызовы выделения и очистки памяти являются довольно тяжёлыми. В некоторых случаях от них можно избавиться:

```
void foo(void) {
    int     i           // Регистр
    int     j           // Стек
    int   * .str.a;     // Регистр
    float .str.b[100]; // Стек

    .str.a = &j;
    // ...
}
```

Память на стеке выделяется и очищается автоматически, а значит у нас отсутствуют накладные расходы на вызовы `malloc` и `free`.

Сверхоперативная память

Для ускорения работы с памятью была сделана иерархия сверхоперативных памятей. Существует несколько видов таких памятей:

- Кэш данных (D-cache)
- Кэш инструкций (I-cache)
- Буфер ассоциативной трансляции (TLB) — информация о трансляции виртуальных адресов в физические.

Компилятор обычно оптимизирует работу с первыми двумя видами кэшей.

Сверхоперативная память

Важным для оптимизации понятием являются локальности:

- **Пространственная** — за небольшой промежуток времени происходит обращение к данным, лежащим рядом.
- **Временная** — использование данных несколько раз за короткий период времени.

Сверхоперативная память

Локальности данных:

- **Пространственная** — последовательное обращение к элементам массива.
- **Временная** — обращение по одному адресу (например глобальной переменной) каждую итерацию цикла

Локальности инструкций:

- **Пространственная** — расположение линейных участков друг за другом.
- **Временная** — исполнение линейного участка в цикле.

Примеры оптимизация работы с памятью

Представим что длина одной строчки L1 кэша равна 32Б. Тогда для цикла будут справедливы следующие характеристики:

```
for(int i = 0; i < N; i++)
    S += a[i];
```

Если `sizeof(a[i]) == 1`, то на 31 попадание имеем 1 промах
Если `sizeof(a[i]) == 8`, то на 3 попадания имеем 1 промах

Примеры оптимизация работы с памятью

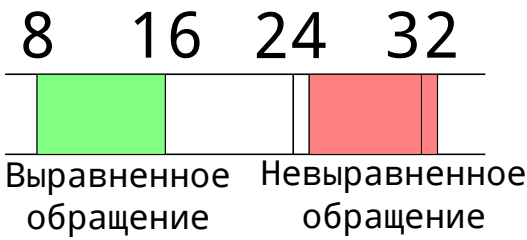
Для уменьшения количества кэш-промахов существует возможность расстановки операций **предподкачки** (prefetch). Это обращение в память, задача которого заранее загрузить данные в кэш:

```
for(int i = 0; i < N; i++)
{
    prefetch(&a[i + x]);
    S += a[i];
}
```

На предподкачку направляются данные, которые будут использованы через несколько итераций. Конкретное смещение зависит от характеристик целевой машины.

Примеры оптимизация работы с памятью

Важным моментом при работе с памятью является выравнивание данных в памяти.



Обращение к памяти будет **выравненным** если его адрес кратен степени двойки.

Примеры оптимизация работы с памятью

Невыравненная память будет создавать дополнительный кэш-промах при обращении к значению на границе строки кэша.

Выравненная память позволяет проводить весомые оптимизации, такие как векторизация и асинхронная подкачка данных (для Эльбруса).

Часть 3.
Секция 4. Оптимизация циклов.

Оптимизация циклов

Существуют разные виды нагрузки на приложения в зависимости от их типа. Для приложений, занимающихся научными и инженерными расчётами характерным признаком является вычислительный цикл, работающий большую часть времени.

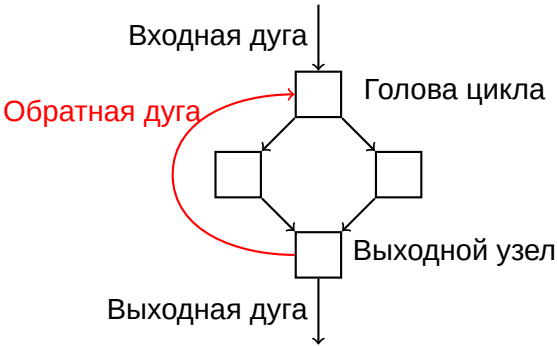
Оптимизации циклов для таких приложений дают наибольший прирост по сравнению с остальными приложениями.

Чем проще организован цикл, тем проще компилятору его оптимизировать.

Структура цикла

Цикл всегда будет иметь:

- Одну голову
- Хотя бы одну обратную дугу — дугу, ведущую из тела цикла в голову
- Хотя бы одну выходную дугу



Структура цикла

Если цикл имеет только одну обратную дугу, одну выходную дугу и входные дуги, входящие только в голову, то такой цикл называется естественным.

Количество входных, обратных и выходных дуг у цикла не ограничено. При этом голова всегда будет только одна. Чем больше различных дуг у цикла тем сложнее его оптимизировать.

Если у цикла есть входная дуга, смотрящая не в голову, то она называется **боковым входом**. Цикл, обладающий боковым входом называется **несводимым**. Такие циклы практически не оптимизируются.

Структура цикла

Операция в цикле называется **рекуррентной** если от её результата зависит операция на следующей итерации.

Высота рекуррентности — минимальное время работы одной физической итерации цикла.

Цикл часто обладает **индуктивной переменной** — переменной, модифицирующейся каждую итерацию и служащей критерием завершения цикла.

Примеры оптимизации циклов

Часть цикловых оптимизаций нацелена на устранение избыточности.
Например **вынос инвариантов** из цикла:

```
for(i = 0; i < M; i++)
    for(j = 0; j < N; j++)
        a[i][j] = b[i][j];
```

$\rightarrow$

```
for(i = 0; i < M; i++)
{
    .a_i = &a[i];
    .b_i = &b[i];
    for(j = 0; j < N; j++)
        .a_i[j] = .b_i[j];
}
```

Примеры оптимизации циклов

Некоторые оптимизации циклов направлены на улучшение работы с памятью, например оптимизация **перестановка циклов** нацелена на увеличение пространственной локальности кэша:

```
for(i = 0; i < M; i++)                for(j = 0; j < N; j++)
    for(j = 0; j < N; j++)     →          for(i = 0; i < M; i++)
        a[i] += b[j][i]                       a[i] += b[j][i]
```

Примеры оптимизации циклов

Очень эффективными оптимизациями являются оптимизации, позволяющие наращивать параллелизм вычислений в цикле. Самой известной такой оптимизацией является **раскрутка цикла**:

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

$\rightarrow$

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

for(i -= 2; i < N; i++)
    a[i] = b[i] + c[i];
```

Примеры оптимизации циклов

Существует и обратная оптимизация — **скрутка цикла**. Она определяет цикл, который был раскручен до неё и приводит его в изначальный вид.

Оптимизация скрутки применяется для случаев когда цикл был неэффективно раскручен человеком и в таком виде будет только ухудшать производительность.

Сама по себе раскрутка (как и большинство других цикловых оптимизаций) обычно применяется не в чистом виде, а в комбинации с другими оптимизациями, например со **слиянием циклов**.

Часть 3.
Секция 5. Межпроцедурные оптимизации.

Примеры межпроцедурных оптимизаций

Вызовы функций для оптимизатора являются проблемными операциями, т.к. часто функция представляет из себя «чёрный ящик», поведение которого неизвестно.

Задачей межпроцедурных оптимизаций является уменьшение накладных расходов от вызова функций либо устранение самих вызовов.

Примеры межпроцедурных оптимизаций

Подстановка функций (inline) — одна из самых важных оптимизаций для получения производительности.

```
int add(int a, int b)          int add(int a, int b)
{return a + b;}                {return a + b;}

                          →

void incr(int * a)             void incr(int * a)
{*a = add(*a , 1);}            {*a = *a + 1;}
```

Примеры межпроцедурных оптимизаций

Эффекты от подстановки функций:

- Устраняет переход и инициализацию функции.
- Устраняет операции пересылки для подготовки аргументов.
- Позволяет смешивать код вызываемой функции с кодом на месте вызова.
- Создаёт большое количество возможностей для других оптимизаций.

Могут наблюдаться и негативные эффекты:

- Увеличивается нагрузка на кэш инструкций
- Может увеличивать размер кода (часто наоборот его уменьшает)
- Значительно увеличивает время компиляции

Примеры межпроцедурных оптимизаций

В некоторых случаях подставить функцию уже не удаётся. При наличии констант в аргументах функции можно сделать копию функции и частично вычислить её тело. Такая оптимизация называется **клонированием функций**:

```
int add(int a, int b)
{return a + b;}

void incr(int * a)
{*a = add(*a ,1);}
```

→

```
int add(int a, int b)
{return a + b;}

int .add1(int a)
{return a + 1;}

void incr(int * a)
{*a = .add1(*a);}
```

Примеры межпроцедурных оптимизаций

При использовании виртуальных методов C++ (или просто вызовов функций по указателю в Си), выполнить подстановку нельзя. Для решения этой проблемы существует класс оптимизаций **подстановки неявного вызова**. Одним из методов этих оптимизаций является **девиртуализация**:

```
class A                          class A
{                                {
public:                          public:
    virtual void foo(){}             virtual void foo(){}
}                                }
                          →
...                              ...
A a;                             A a;
// Вызов по указателю            // Вызов по имени
a.foo();                         A::foo(&a);
```

Заключение.

Заключение

На текущий момент самой сложной и долго работающей частью компилятора является оптимизатор.

Важным фактором оптимизации является возможность увидеть как можно больше инструкций в рамках одного линейного участка для обеспечения эффективного планирования.

Оптимизации становятся наиболее эффективны при грамотном выборе последовательности их применения.

Не любой код может быть оптимизирован. Программист должен понимать что может помешать компилятору работать и что может помочь ему при выполнении оптимизаций.