11. Открытая вставка

Встраивание функций (inlining)

```
class Bar {
  int size;
  Bar(int _size) {
    size = _size;
  }
  int getSize() {
    return size;
  }
}
Bar a(10), b(20);
int x = a.getSize() +
    b.getSize();
```



```
int sum = a.size + b.size;
```



```
int sum = 30;
```

- Встраивание особенно эффективно для C++ кода, где часто используются небольшие функции для доступа к полям классов

Встраивание функций (inlining)

```
int foo(int a, int b, int c, int d,
        int e)
{
  return a + b + c + d + e;
}

int main() {
  return foo(1, 2, 3, 4, 5);
}
```



```
int main() {
  return 1 + 2 + 3 + 4 + 5;
}
```

- Увеличится ли размер кода после выполнения оптимизации? Станет ли быстрее? Что нужно учитывать, чтобы вычислить это?
- По умолчанию, в GCC встраивание выполняется, только если в результате размер кода уменьшится.

Встраивание функций (inlining)

```
void bar();

int foo(int a, int b, int c, int d,
        int e)
{
  int arr[100];
  …
  bar();
  return a + b + c + d + e;
}

int main() {
  return foo(1, 2, 3, 4, 5);
}
```

Пролог:
- Сохранение регистров, которые «портит» функция
- Сохранение и установка stack pointer

Эпилог:
- Восстановление регистров и stack pointer
- Возврат из функции

Вызов функции:
- Подготовка аргументов

```
foo:
        .cfi_startproc
Пролог  movq    %rbx, -32(%rsp)
        movq    %rbp, -24(%rsp)
        movq    %r12, -16(%rsp)
        movq    %r13, -8(%rsp)
        subq    $440, %rsp
        ...
Эпилог  movq    408(%rsp), %rbx
        movq    416(%rsp), %rbp
        movq    424(%rsp), %r12
        movq    432(%rsp), %r13
        addq    $440, %rsp
        ret

main:
        ...
Вызов   movl    $5, %r8d
функции movl    $4, %ecx
        movl    $3, %edx
        movl    $2, %esi
        movl    $1, %edi
        call    foo
```


x86_64

Встраивание функций (inlining)

```
void bar();

int foo(int a, int b, int c, int d,
        int e)
{
  int arr[100];
  …
  bar();
  return a + b + c + d + e;
}

int main() {
  return foo(1, 2, 3, 4, 5);
}
```

Пролог:

- Сохранение регистров, которые «портит» функция, а также адреса возврата (в **lr**)
- Сохранение и установка stack pointer

Эпилог:

- Восстановление регистров и stack pointer
- Возврат из функции (lr записывается в pc)

```
Пролог:
foo:
        stmfd   sp!, {r4, r5, r6, lr}
        sub     sp, sp, #400
        …
        ldr     r4, [sp, #416]
        …
Эпилог:
        add     sp, sp, #400
        ldmfd   sp!, {r4, r5, r6, pc}
        bx      lr

main:
        …
Вызов функции:
        mov     r3, #5
        str     r3, [sp]
        mov     r0, #1
        mov     r1, #2
        mov     r2, #3
        mov     r3, #4
        bl      foo
```

ARMv7

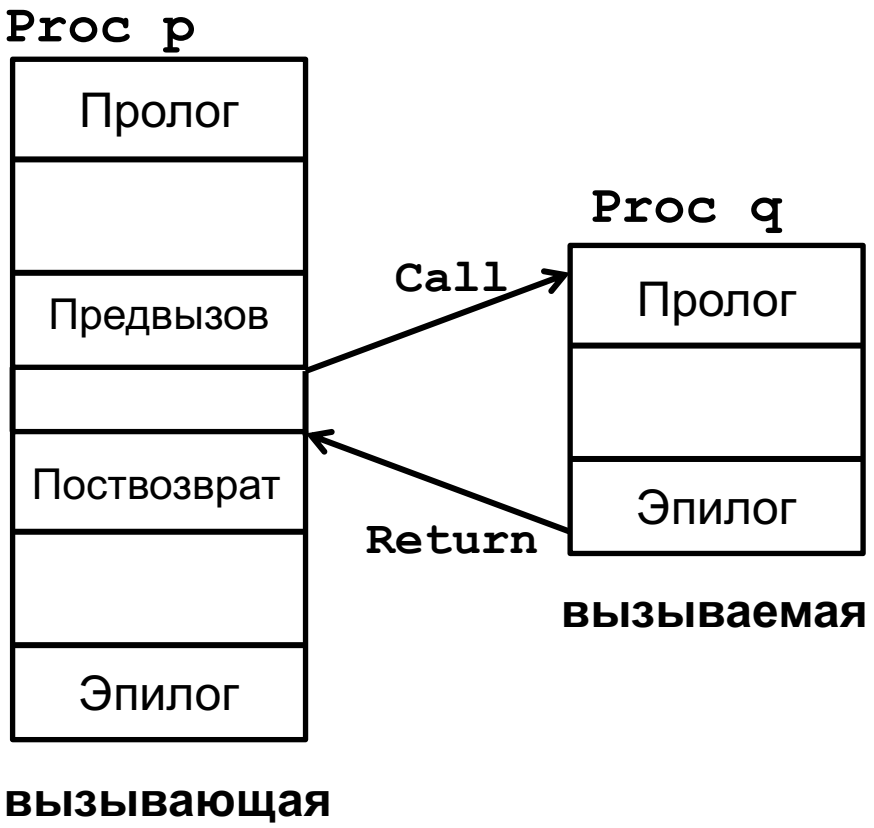
Вызов функции:

- Подготовка 1-х четырех аргументов в регистры, 5-го в стек

10.1 Постановка задачи

10.1.1 Вызов процедуры

- ◊ На схеме показана типичная реализация вызова процедуры **q** из процедуры **p**.
- ◊ Процедура может иметь несколько *точек вызова.*
- ◊ Общее правило: перенос операторов из "Предвызова" и "Поствозврата" вызывающей соответственно в пролог и эпилог вызываемой позволяет сократить общий размер результирующего кода, так как в случае нескольких вызовов одной и той же процедуры приводит к замене нескольких операций копирования в предвызовах одной операцией копирования в прологе.



10.1 Постановка задачи

10.1.1 Вызов процедуры

◊ **Предвызов**:

1. Вычисление фактических параметров
2. Если параметр, вызываемый по ссылке, находится на регистре, он загружается в память вызывающей и передает соответствующий адрес вызываемой
3. Определение адреса возврата и, в случае необходимости адресов переменных для приема возвращаемых значений.
   Если параметров меньше k, они передаются на регистрах.
   Если параметров больше k, первые k параметров передаются на регистрах, а остальные помещаются в *буфер обмена*, расположенный либо в памяти вызываемой, либо в памяти вызывающей.
4. Сохраняются caller-saved регистры.

◊ **Поствозврат**:

1. Восстанавливает caller-saved регистры.
2. Запоминает значения, возвращенные на регистрах.

10.1 Постановка задачи

10.1.1 Вызов процедуры

◊ **Пролог**:

1. Выделяет память для хранения значений фактических параметров, передаваемых вызывающей на регистрах, а также для локальных переменных вызываемой.
2. Размещает и инициализирует локальные переменные.
3. Сохраняет callee-saved регистры.

◊ **Эпилог**:

1. Выделяет память для возвращаемых значений.
2. Размещает возвращаемые значения.
3. Восстанавливает callee-saved регистры.
4. Передает управление по адресу возврата.

Предвызов, поствозврат, пролог и эпилог – **накладные расходы на вызов процедуры**

10.1 Постановка задачи

10.1.1 Вызов процедуры

◊ **Пролог**:

1. Выделяет память для хранения значений фактических параметров, передаваемых вызывающей на регистрах, а также для локальных переменных вызываемой.
2. Размещает и инициализирует локальные переменные.
3. Сохраняет callee-saved регистры.

◊ **Эпилог**:

1. Выделяет память для возвращаемых значений.
2. Размещает возвращаемые значения.
3. Восстанавливает callee-saved регистры.
4. Передает управление по адресу возврата.

Предвызов, поствозврат, пролог и эпилог – **накладные расходы на вызов процедуры**

При этом тело самой функции может занимать всего пару строк.

10.1 Постановка задачи

10.1.2 Деление программы на процедуры. Недостатки

- Деление программы на процедуры имеет много достоинств, так как существенно упрощает составление и отладку программ.
- Но такое деление имеет и недостатки, усложняющие оптимизацию программ во время компиляции:
  - **Накладные расходы на вызов процедур** (прологи, эпилоги и т.п.) существенно снижают эффективность (быстродействие) программы.
  - **Ограничение возможностей компилятора** понимать, что происходит внутри вызова.
    Рассмотрим фрагмент процедуры

```
x = 15;
p = &x;
m = n + x;
q = f(p, m + n);
m = n + x;
```

Вопрос: можно ли исключить одно из присваиваний `m = n + x;`?

10.1 Постановка задачи

10.1.2 Деление программы на процедуры. Недостатки

- Деление программы на процедуры имеет много достоинств, так как существенно упрощает составление и отладку программ.
- Но такое деление имеет и недостатки, усложняющие оптимизацию программ во время компиляции:
  - **Накладные расходы на вызов процедур** (прологи
  - Для преодоления первого недостатка используется **открытая вставка**.
  - **Ограничение возможностей компилятора** понимать, что происходит внутри вызова.
    Рассмотрим, например, фрагмент процедуры

```
x = 15;
p = &x;
m = n + x;
q = f(p, m + n);
m = n + x;
```

Вопрос: можно ли исключить одно из присваиваний `m = n + x;`?

10.1 Постановка задачи

10.1.2 Деление программы на процедуры. Недостатки

- Деление программы на процедуры имеет много достоинств, так как существенно упрощает составление и отладку программ.
- Но такое деление имеет и недостатки, усложняющие оптимизацию программ во время компиляции:
  - **Накладные расходы на вызов процедур** (прологи
- Для преодоления первого недостатка используется **открытая вставка**.
  - **Ограничение возможностей компилятора** понимать, что происходит внутри вызова.
    Рассмотрим фрагмент процедуры **g**

```
x = 15;
```

- Для преодоления второго недостатка используется межпроцедурный анализ (совместный анализ процедур **g** и **f**).
- Открытая вставка также позволяет устранить второй недостаток, после того как код обеих функций оказывается в одной.

Вопрос: можно ли исключить одно из присваивании **m = n + x;**?

10.2 Открытая вставка

10.2.1 Что это такое

- ◊ Вызовы процедур в среднем составляют примерно 5% всех инструкций программы, причем вызов процедуры – самый дорогой оператор программы
- ◊ Оптимизация состоит в том, что **вызов процедуры исключается из программы и вместо него в программу вставляется тело вызываемой процедуры**, а также две группы операторов присваивания, чтобы совместить контекст вызываемой процедуры с контекстом вызывающей.

10.2 Открытая вставка

10.2.2 Как выполняется открытая вставка

- Открытая вставка это очень просто:
  - заменить вызов на тело вызываемой процедуры
  - превратить передачу параметров и возврат результатов в инструкции присваивания
  - выполнить оптимизацию "распространение копий", чтобы исключить лишние переменные
  - правильно разобраться с областями видимости переменных, переименовывая переменные в случае необходимости (для программ в промежуточном представлении MIR это не требуется)

10.2 Открытая вставка

10.2.3 Преимущества и недостатки открытой вставки

- **Преимущества:**
  - исключаются накладные расходы на организацию вызова процедуры и возврата из нее
  - исключаются накладные расходы на передачу параметров и возврат результатов
  - появляется возможность оптимизации вызываемой процедуры с учетом контекста вызывающей (открытая вставка как один из способов реализации межпроцедурного анализа)
- **Недостатки:**
  - могут возрасти требования скомпилированного кода по памяти
  - может замедлиться компиляция
  - сложности с рекурсией

10.2 Открытая вставка

10.2.4 Виртуальная открытая вставка

◊ Виртуальная открытая вставка: вставка моделируется (симулируется) во время анализа вызывающей процедуры, **но фактически не выполняется**.

10.2 Открытая вставка

10.2.4 Виртуальная открытая вставка

◊ Виртуальная открытая вставка: вставка моделируется (симулируется) во время анализа вызывающей процедуры, **но фактически не выполняется**.

◊ Ответ на вопрос: "Как изменится процедура **P**, если в нее будет вставлена процедура **A**?" и другие схожие вопросы.

10.2 Открытая вставка

10.2.5 Какие вызовы следует заменять открытой вставкой?

- Как принимаются решения, какие вызовы лучше заменить открытой вставкой:
  - размеры вызывающей и вызываемой процедур (легко вычислить размеры **до** открытой вставки а что можно сказать о размерах **после** открытой вставки?)
  - частота вызова (статические оценки или динамическое профилирование)
  - вызовы, при которых вызываемая процедура существенно улучшается после совместной оптимизации (неясно, как это оценить численно)
  - **аннотации** (наборы соотношений)
    - вызываемой процедуры (в начале вызываемой процедуры)
    - контекста в точке вызова вызываемой процедуры

10.2 Открытая вставка

10.2.5 Какие вызовы следует заменять открытой вставкой?

◊ Как принимаются решения, какие вызовы лучше заменить открытой вставкой:

- ◈ размеры вызывающей и вызываемой процедур
- ◈ **аннотации** (наборы соотношений)
  - ♦ вызываемой процедуры (в начале вызываемой процедуры)
  - ♦ контекста в точке вызова вызываемой процедуры

Что такое профилирование? *Профилирование* – это численная оценка времени выполнения каждого фрагмента процедуры (например, базового блока, области, всей процедуры).

В результате профилирования получается *временной*, или *частотный*, или еще какой-нибудь *профиль* процедуры.

10.2 Открытая вставка

10.2.5 Какие вызовы следует заменять открытой вставкой?

◊ Как принимаются решения, какие вызовы лучше заменить открытой вставкой:

Как получить профиль? Два метода: инструментирование и семплирование.

а что можно сказать о размерах **после** открытой вставки?)

Инструментирование – в различные точки процедуры вставляются вызовы соответствующего инструмента (счетчика количества обращений к фрагменту, секундомера и т.п.), выбираются наборы исходных данных, обеспечивающие достаточно полное покрытие процедуры и исследуемая процедура запускается на этих наборах, в результате чего получается требуемый профиль.

- вызываемой процедуры (в начале вызываемой процедуры)
- контекста в точке вызова вызываемой процедуры

10.2 Открытая вставка

10.2.5 Какие вызовы следует заменять открытой вставкой?

◊ Как принимаются решения, какие вызовы лучше заменить открытой вставкой:

Как получить профиль? Два метода: инструментирование и семплирование.

а что можно сказать о размерах **после** открытой вставки?)

Семплирование – во время выполнения процедуры профилировщик периодически спрашивает у VM (или у ОС) как выглядят трассы всех потоков и, получив ответ, соответственно обновляет свою статистику. Это, как правило меньше замедляет профилируемую программу, но дает менее точный профиль.

- ◊ **аннотации** (наборы соотношений)
  - ♦ вызываемой процедуры (в начале вызываемой процедуры)
  - ♦ контекста в точке вызова вызываемой процедуры

10.2 Открытая вставка

10.2.5 Какие вызовы следует заменять открытой вставкой?

◊ Как принимаются решения, какие вызовы лучше заменить открытой вставкой:

Как получить профиль? Два метода: инструментирование и семплирование.

а что можно сказать о размерах **после** открытой вставки?)

Семплирование – во время выполнения процедуры профилировщик периодически спрашивает у VM (или у ОС) как выглядят трассы всех потоков и, получив ответ, соответственно обновляет свою статистику. Это, как правил ...му, но дает менее точ...

How to Profile a C program in Linux using GNU gprof
(https://www.maketecheasier.com/profile-c-program-linux-using-gprof/)

◊ **а**

- ...ваемой процедуры)
- контекста в точке вызова вызываемой процедуры

10.2 Открытая вставка

10.2.5 Какие вызовы следует заменять открытой вставкой?

◊ Как принимаются решения, какие вызовы лучше заменить открытой вставкой:

Как получить профиль? Два метода: инструментирование и семплирование.

(а что можно сказать о размерах **после** открытой вставки?)

Семплирование – во время выполнения процедуры профилировщик периодически спрашивает у VM (или у ОС) как выглядят трассы всех потоков и, получив ответ, соответственно обновляет свою статистику. Это, как правил ...му, но дает менее

How to Profile a C program in Linux

◊ ...gram-

Средство профилирования **Windows** называется **Performance Counters** (см. сайт https://docs.microsoft.com/en-us/windows/desktop/PerfCtrs/about-performance-counters)

...ваемой

...емой процедуры

10.3 Открытая вставка при компиляции С-программ

10.3.1 Когда следует выполнять открытую вставку

◊ Открытую вставку нужно выполнять прежде остальных оптимизаций, таких как сворачивание констант и исключение мертвого кода. Поэтому естественно выполнять открытую вставку во время компиляции.

◊ Открытая вставка может выполняться как на фазе построения АСД (в рамках переднего плана), так и над программой, представленной в машинно-независимом трехадресном коде (в виде последовательности "четверок").

◊ Открытая вставка требует знания характеристик всех вызовов функций, включая библиотечные, что ограничивает возможности открытой вставки при раздельной компиляции.
Для С-программ последнее замечание ослабляется тем, что единица компиляции (модуль) обычно содержит определения нескольких процедур и объявления всех остальных (даже внешних) процедур, вызываемых в модуле.

10.3 Открытая вставка при компиляции С-программ

10.3.1 Когда следует выполнять открытую вставку

◊ В последнее время открытую вставку нередко выполняют в самом конце процесса компиляции – во время связывания программ, так как на этом этапе доступны вообще все процедуры программы.

◊ Недостатком такого подхода является **необходимость повторной оптимизации программы после вставки**, так как только в этом случае можно получить все выгоды от открытой вставки.

10.3 Открытая вставка при компиляции С-программ

10.3.2 Представление программы для реализации открытой вставки

◊ При открытой вставке используется представление оптимизируемой программы в виде взвешенного графа вызовов.

◊ **Взвешенный граф вызовов** (**ВГВ**) задается пятеркой

$$G = \langle N, p_N, E, p_E, main \rangle$$

N – множество вершин (представляющих процедуры программы), с каждой вершиной связан ее "вес"

p_N – целое число, равное, например, количеству инструкций соответствующей процедуры

E – множество ребер (каждое ребро соединяет вызываемую и вызывающую процедуры), с каждым ребром связан его "вес"

p_E – количество выполнений соответствующего вызова

$main$ – имя процедуры, выполняемой первой.

10.3 Открытая вставка при компиляции С-программ

10.3.2 Представление программы для реализации открытой вставки

◊ При открытой вставке используется представление оптимизируемой программы в виде взвешенного графа вызовов.

◊ **Взвешенный граф вызовов** (**ВГВ**) задается пятеркой

$$G = \langle N, p_N, E, p_E, main \rangle$$

N – множество вершин (представляющих процедуры программы), с каждой вершиной связан ее "вес"

p_N – целое число, равное, например, количеству инструкций соответствующей процедуры

Обычно p_N и p_E определяются во время профилирования

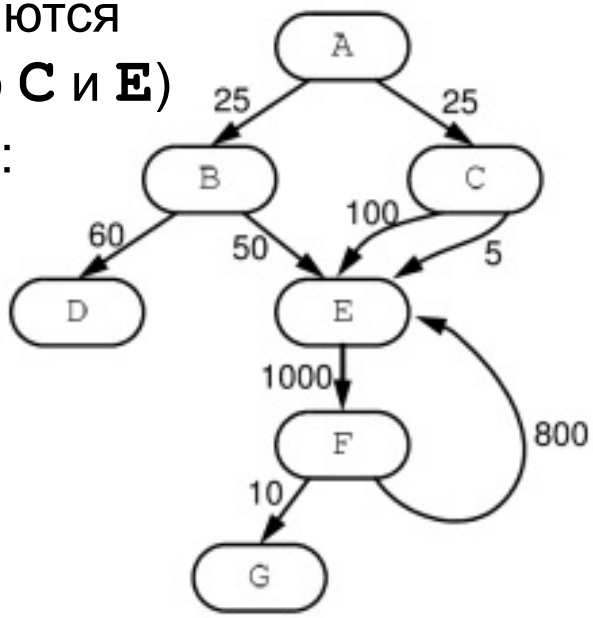
p_E – количество выполнений соответствующего вызова

$main$ – имя процедуры, выполняемой первой.

10.3 Открытая вставка при компиляции С-программ

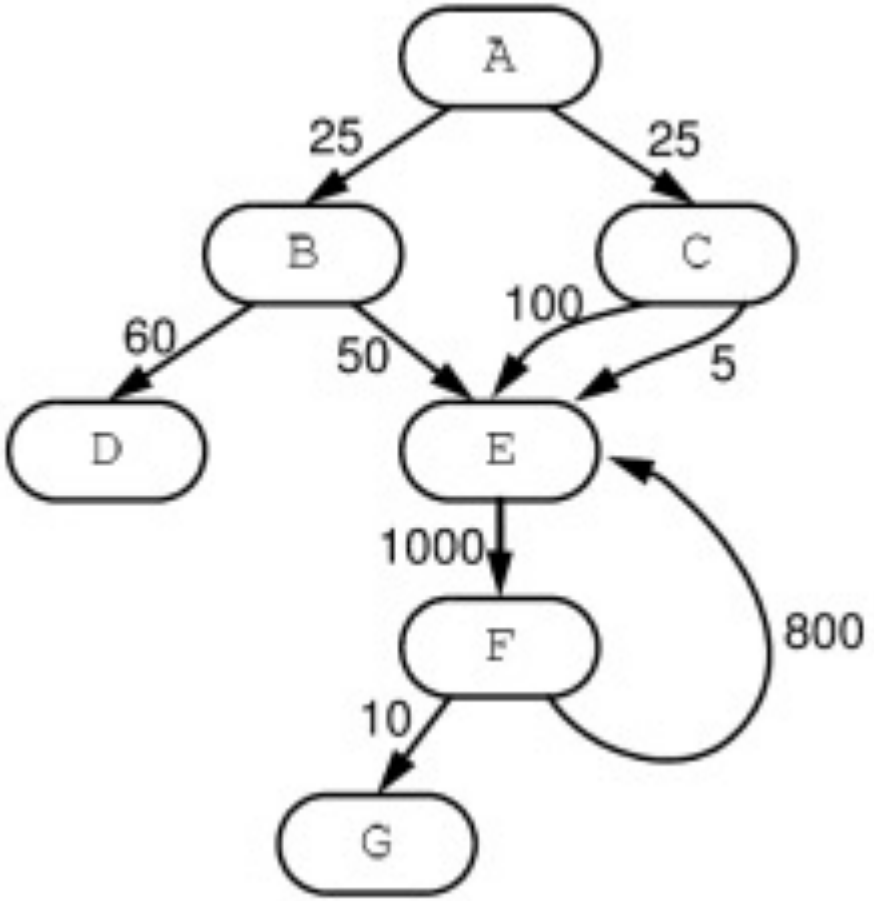
10.3.2 Представление программы для реализации открытой вставки

- Граф вызовов:
  - вершины соответствуют процедурам
  - дуги соответствуют вызовам процедур (дуги помечены числами, означающими либо количество, либо частоту вызовов)
  - если процедура вызывается несколько раз, соответствующие вершины соединяются несколькими дугами (на рисунке это **C** и **E**)
- Трудные случаи построения графа вызовов:
  - вызовы внешних процедур
  - вызовы из внешних процедур
  - вызовы по указателям, значениям функций



10.3 Открытая вставка при компиляции С-программ

10.3.2 Представление программы для реализации открытой вставки



10.3 Открытая вставка при компиляции С-программ

10.3.2 Представление программы для реализации открытой вставки

$$G = \langle N, p_N, E, p_E, main \rangle$$

- Каждая **вершина** ВГВ содержит следующую информацию:
  - тело процедуры
  - вес
  - множество исходящих ребер
- Каждое **ребро** ВГВ содержит следующую информацию:
  - уникальный идентификатор ребра (для задания ребра недостаточно указать пару вершин, так как если вызов происходит несколько раз, существует несколько соответствующих ребер)
  - имя вызывающей процедуры
  - имя вызываемой процедуры
  - вес ребра
  - статус вызываемой процедуры (статус вычисляется в процессе анализа и имеет три значения: анализа не было, вставка запрещена, вставка выполнена)

10.3 Открытая вставка при компиляции С-программ

10.3.2 Представление программы для реализации открытой вставки

$$G = \langle N, p_N, E, p_E, main \rangle$$

- **Веса** вершин и ребер можно определить:
  - либо с помощью структурного анализа ГПУ
  - либо с помощью профилирования
    (мы рассматриваем два способа профилирования – инструментирование и семплирование)

10.3 Открытая вставка при компиляции С-программ

10.3.2 Представление программы для реализации открытой вставки

$$G = \langle N, p_N, E, p_E, main \rangle$$

- ◊ Открытая вставка статического вызова состоит в удалении из ВГВ часто выполняемой дуги с одновременной вставкой тела вызываемой процедуры в тело вызывающей в соответствии с идентификатором дуги.
- ◊ Удаление дуги может привести к тому, что вызываемая процедура станет недостижимой в ВГВ. В этом случае процедура больше нигде не вызывается, и ее нужно удалить.
- ◊ Когда вершина имеет ребро, которое является одновременно входящим и исходящим, имеет место рекурсивный вызов. В этом случае можно попытаться использовать стандартную технику обработки рекурсивных вызовов, заменяя их циклами (если имеет место хвостовая рекурсия), либо выполнить открытую вставку рекурсивного вызова (на глубину 1).

10.3 Открытая вставка при компиляции С-программ

10.3.3 Выбор вызова для замены

- Казалось бы, выполняя открытую вставку, выгодно заменить как можно больше "тяжелых" ребер. К сожалению это не всегда так.
  - Необходимо избегать вставки слишком длинного кода (для этого следует установить верхнюю границу длины вставляемого кода)
  - Вставка кода, использующего слишком много места на стеке вызовов (например, рекурсия) может привести к быстрому переполнению стека. Этого тоже желательно избегать.

10.3 Открытая вставка при компиляции С-программ

10.3.3 Выбор вызова для замены

◊ **Взрывной рост объема кода**

Для того, чтобы развернуть вызов, необходимо сдублировать тело вызываемой процедуры и подставить полученную копию в вызывающую процедуру. Ясно, что в результате общий объем программы возрастает.

◊ Скорее всего, верно, что многие С-функции вызываются только один раз и поэтому копии оригинала этих один раз вызываемых функций можно удалить как недостижимые после их вставки.
К сожалению, так удается удалить не все недостижимые узлы в неполном ВГВ (Большая часть ВГВ больших процедур неполны, так как не могут учесть все системные вызовы).
Поэтому необходимо установить верхнюю границу на размер области инструкций процедуры. Эта граница может быть задана либо как фиксированное целое число, либо как функция от размера процедуры.

10.3 Открытая вставка при компиляции С-программ

10.3.3 Выбор вызова для замены

◊ **Переполнение стека вызовов.**

Стек вызовов поддерживает передачу параметров, сохранение регистров, размещение локальных (автоматических) переменных, выделение памяти под возвращаемое значение, в связи с вызовом процедуры. Для каждой вызываемой процедуры нетрудно подсчитать размер необходимой для ее вызова области стека вызовов.

Переполнение стека вызовов может произойти при вставке рекурсивной процедуры, которой требуется очень большая область для поддержки рекурсивного вызова.

10.3 Открытая вставка при компиляции С-программ

10.3.3 Выбор вызова для замены

◊ **Переполнение стека вызовов.**

Например, пусть определены рекурсивная функция **`m(x)`**

и еще одна функция **`n(x)`** :

```
m(x) {return(x ? m(x-1)+m(x-2)+n(x): 1);}
n(x) {int y[100000];…}
```

Когда **`m(x)`** вызывается для достаточно больших значений **`x`**,

`n(x)` может существенно увеличить размер стека вызовов.

◊ Для предотвращения переполнения стека вызовов необходимо ввести фиксированный предельный размер области стека вызовов, необходимой для поддержки рекурсивного вызова.

10.3 Открытая вставка при компиляции С-программ

10.3.3 Выбор вызова для замены

◊ **Функция затрат**

Для заданного ВГВ функция затрат (ФЗ) предназначена для выявления наиболее подходящего ребра для открытой вставки. ФЗ позволяет сформулировать открытую вставку как оптимизационную задачу минимизации затрат на открытую вставку для произвольного количества вставок.

◊ Ввиду того, что ни реальные затраты на вставку, ни выгода от вставки неизвестны во время компиляции, получить оптимальные решения невозможно. Более того, пространство поиска для сформулированной оптимизационной задачи слишком велико для практической реализации.

Поэтому желательно использовать эвристику, позволяющую существенно сохранять небольшие размеры пространства поиска.

◊ При разработке функции затрат необходимо не только предотвратить рассмотренные опасности, связанные с переполнением, но и исключить из ВГВ несущественные ребра.

10.3 Открытая вставка при компиляции С-программ

10.3.3 Выбор вызова для замены

◊ **Пример функции затрат**

```
cost(G, arc Ai) =
  if((caller is recursive)&&
     (control_stack_usage(Ai)>BOUND))
  then cost = INFINITY;                    // ∞
  else
     if(weight(Ai) < MIN)
     then cost = INFINITY;
     else
       if(instruction_space_after_expansion
          (G,Ai) > MAX)
       then cost = INFINITY;
       else
          cost = Code_expansion_Cost of Ai -
                    Benefits_of_Inlining Ai;
```

10.3 Открытая вставка при компиляции С-программ

10.3.3 Выбор вызова для замены

◊ **Пример функции затрат**

```
cost(G, arc Ai) =
  if((caller is recursive)&&
     (control_stack_usage(Ai)>BOUND))
  then cost = INFINITY;                    // ∞
  else
     if
     then cost = INFINITY;
     else
        (G,Ai) > Ai;
        then cost = INFINITY;
        else
           cost = Code_expansion_Cost of Ai -
                     Benefits_of_Inlining Ai;
```

INFINITY означает, что затарты на вставку непомерно высоки, так что вставка нецелесообразна

Code expansion Cost – затраты на вставку

Benefits_of_Inlining – выгоды от вставки

10.3 Открытая вставка при компиляции С-программ

10.3.3 Выбор вызова для замены

◊ **Функция затрат**

Затраты на вставку кода – это увеличение памяти, необходимой для размещения программы, и влияние на производительность кэш-памяти для инструкций. Точные значения этих величин не могут быть получены во время компиляции.
Грубая оценка затрат может быть получена путем умножения эмпирической константы на размер кода.

10.3 Открытая вставка при компиляции С-программ

10.3.4. Вставка функции

◊ После того как определены функции для вставки и места, в которые их нужно вставить, выполняется **фаза физической вставки функций**.

Эта фаза состоит в решении трех основных задач:

1) дублирование вызываемой функции
2) переименование переменных
3) модификация таблицы символов.

◊ Выполнение дублирования тривиально. Однако при его реализации бывает необходимо выполнять такие оптимизации реализуемого алгоритма как кэширование определений наиболее часто вставляемых функций, чтобы сократить количество чтений файлов.

10.3 Открытая вставка при компиляции С-программ

10.3.4. Вставка функции

◊ После того как определены функции для вставки и места, в которые их нужно вставить, выполняется **фаза физической вставки функций**.

Эта фаза состоит в решении трех основных задач:

1) дублирование вызываемой функции
2) переименование переменных
3) модификация таблицы символов.

◊ Выполнение дублирования тривиально. Однако при его реализации бывает необходимо выполнять такие оптимизации реализуемого алгоритма как кэширование определений наиболее часто вставляемых функций, чтобы сократить количество чтений файлов.

Дублирование вставляемой функции необходимо, так как ее могут вызывать и функции, в которые она не вставлена.

10.3 Открытая вставка при компиляции С-программ

10.3.4. Вставка функции

◊ Переименование формальных параметров и локальных переменных должно выполняться на новой копии вызываемой функции перед ее вставкой в вызывающую.
Если вставка выполняется на уровне исходного кода, для упрощения задачи можно вводить новые области видимости локальных переменных, так что переименовывать придется только формальные параметры.
Можно ввести новые временные переменные для буферизации значений фактических параметров и заменять формальные параметры на эти временные переменные.
На последней стадии лишние инструкции можно исключить с помощью таких оптимизаций как распространение копий, исключение мертвого кода, исключение избыточных вычислений и др.

10.3 Открытая вставка при компиляции С-программ

10.3.5. Выяснение недостающей информации

◊ Когда для некоторых функций компилятор или линкер не может получить доступ к вызову функции и характеристикам использования в ней переменных, граф вызовов является неполным.
Как правило это происходит потому, что определения указанных функций и переменных содержатся в модулях программы или в библиотеках, к которым нет доступа. Такие функции называются *внешними функциями*.

◊ Другим источником неопределенностей в графе вызовов являются функции, вызываемые по указателю (вызов по указателю – отличительная черта языка С).

10.3 Открытая вставка при компиляции С-программ

10.3.5. Выяснение недостающей информации

◊ Для вызовов внешних функций и вызовов по указателю имеет смысл ввести специальный тип ребер графа вызовов.

◊ Если в случае вызова по указателю нет возможности решить, какие функции реально могут быть вызваны приходится предполагать наихудший возможный вариант, когда может быть вызвана каждая функция. Такое предположение приводит к множеству дополнительных ребер и циклов в графе вызовов

10.3 Открытая вставка при компиляции С-программ

10.3.6. Исключение недостижимых функций

◊ Выполнение программы всегда начинается с функции **`main`**, поэтому любая функция, недостижимая из функции **`main`**, никогда не будет выполняться, и ее следует исключить.

◊ Функция **`f()`** *достижима* из **`main`**, если

- ◈ на ВГВ имеется путь из **`main`** в **`f()`** или
- ◈ **`f()`** может быть вызвана обработчиком исключительных ситуаций или может быть активизирована каким-нибудь другим асинхронным событием.

◊ Для С-программ такие функции можно обнаружить, идентифицируя все функции, адреса которых используются в выполняемой программе.

◊ Кроме того, мы вынуждены считать (консервативность), что все внешние функции достижимы.

10.4 Эвристики

10.4.1 Простейшая стратегия

- Стратегия 1: Поверхностный анализ
  - изучение исходного кода вызываемой функции, чтобы оценить затраты
  - использование оценок затрат для решения, когда выполнять вставку
  - оптимизация программы после вставки, чтобы удалить лишние переменные (распространение копий)

10.4 Эвристики

10.4.2 Стратегия 2

- Стратегия 2: Глубокий анализ
  - выполнить вставку
  - выполнить после вставки анализ и оптимизацию полученной программы
  - оценить улучшения программы от оптимизаций и измерить объем кода после оптимизаций
  - отказаться от вставки, если сумма затрат превышает суммарные улучшения
  - намного увеличивает время компиляции

10.4 Эвристики

10.4.3 Стратегии 3 и 4

- Стратегия 3: Эвристики (модифицированная версия стратегии 2)
  - выполнить стратегию 2: "пробную" вставку
  - записать компромиссы между затратами и выгодами в постоянную базу данных
  - использовать предыдущие результаты по компромиссам для "похожих" точек вызова

- Стратегия 4: Использование методов машинного обучения