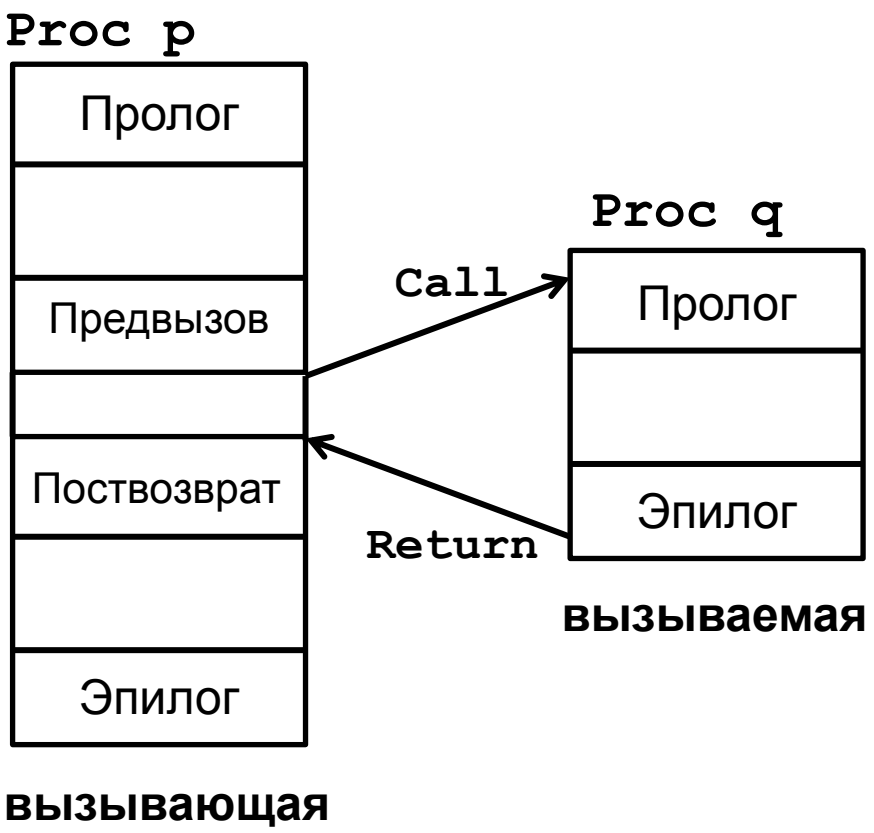
11. Открытая вставка

11.1 Постановка задачи

11.1.1 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.



11.1 Постановка задачи

11.1.1 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.

◊ **Поствозврат**:

1. Запоминает значения, возвращенные на регистрах.
2. В случае необходимости освобождает динамическую память вызываемой.

11.1 Постановка задачи

11.1.1 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.

◊ **Эпилог**:

1. Выделяет память для возвращаемых значений.
2. Размещает возвращаемые значения.
3. Передает управление по адресу возврата.

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

11.1 Постановка задачи

11.1.1 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.

◊ **Эпилог**:

1. Выделяет память для возвращаемых значений.
2. Размещает возвращаемые значения.
3. Передает управление по адресу возврата.

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

А тело процедуры **`sin(x)`** занимает 4 строчки.

11.1 Постановка задачи

11.1.2 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;`?

11.1 Постановка задачи

11.1.2 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**.
  - (эвристическое) программы.
  - ◊ **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;`?

11.1 Постановка задачи

11.1.2 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**).

```
m = n + x;
```

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

11.2 Открытая вставка

11.2.1 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.

11.2 Открытая вставка

11.2.2 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)

11.2 Открытая вставка

11.2.3 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

11.2 Открытая вставка

11.2.4 Виртуальная открытая вставка

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

11.2 Открытая вставка

11.2.4 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.

11.2 Открытая вставка

11.2.5 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

11.2 Открытая вставка

11.2.5 Какие вызовы следует заменять открытой вставкой?

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

- ◈ размеры вызывающей и вызываемой процедур
- ◈ улучшается после совместной оптимизации (непонятно, как это оценить численно)
- ◈ **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.

11.2 Открытая вставка

11.2.5 Какие вызовы следует заменять открытой вставкой?

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

Как получить профиль? Два метода: инструментирование и семплирование.

(можно вычислить размеры до открытой вставки,
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

11.2 Открытая вставка

11.2.5 Какие вызовы следует заменять открытой вставкой?

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

Как получить профиль? Два метода: инструментирование и семплирование.

(можно вычислить размеры до открытой вставки,
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

11.2 Открытая вставка

11.2.5 Какие вызовы следует заменять открытой вставкой?

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

Как получить профиль? Два метода: инструментирование и семплирование.

(можно вычислить размеры до открытой вставки
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

11.2 Открытая вставка

11.2.5 Какие вызовы следует заменять открытой вставкой?

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

Как получить профиль? Два метода: инструментирование и семплирование.

(…что вы знаете о размерах до открытой вставки … 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-…ваемой …емой процедуры

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

11.3 Открытая вставка при компиляции С-программ

11.3.1 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.

11.3 Открытая вставка при компиляции С-программ

11.3.1 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.

11.3 Открытая вставка при компиляции С-программ

11.3.2 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.

11.3 Открытая вставка при компиляции С-программ

11.3.2 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

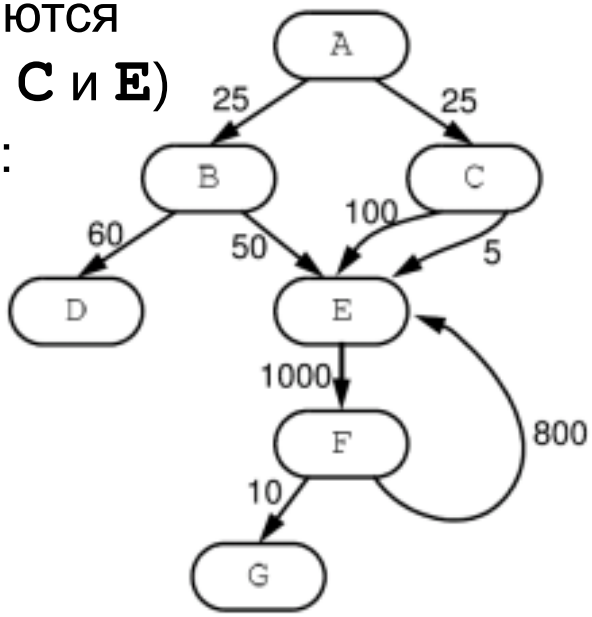
p_E – количество выполнений соответствующего вызова

$main$ – имя процедуры, выполняемой первой.

11.3 Открытая вставка при компиляции С-программ

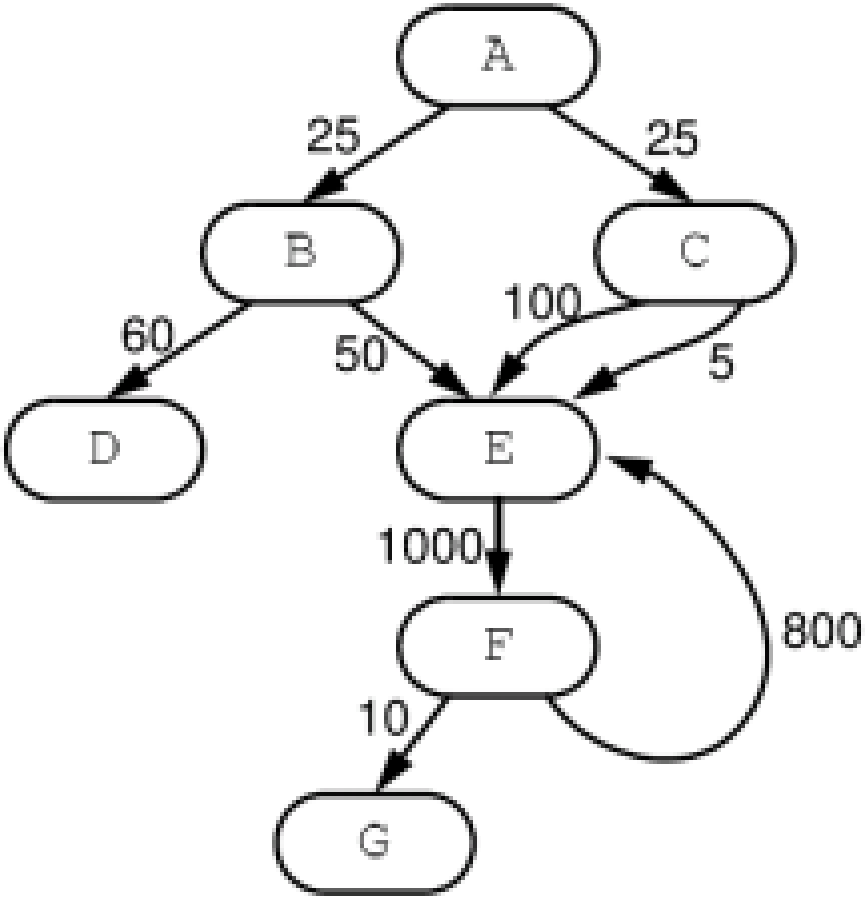
11.3.2 Представление программы для реализации открытой вставки

- 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



11.3 Открытая вставка при компиляции С-программ

11.3.2 Представление программы для реализации открытой вставки



11.3 Открытая вставка при компиляции С-программ

11.3.2 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)

11.3 Открытая вставка при компиляции С-программ

11.3.2 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)
- Войти в каждую вершину ВГВ можно через любое из входящих в нее ребер, поэтому необходимо знать веса всех исходящих ребер, соответствующих каждому входящему ребру.
  В этом случае после выполнения открытой вставки веса ребер остаются правильными.
- Будет показано, что для эффективного выполнения открытой вставки необходимо иметь полный набор точных весов всех вершин и ребер ВГВ

11.3 Открытая вставка при компиляции С-программ

11.3.2 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 мы вставили мертвый код, и его нужно удалить
- ◊ Когда вершина имеет ребро, которое является одновременно входящим и исходящим, имеет место рекурсивный вызов. В этом случае следует использовать стандартную технику обработки рекурсивных вызовов, заменяя их циклами.

11.3 Открытая вставка при компиляции С-программ

11.3.3 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.

11.3 Открытая вставка при компиляции С-программ

11.3.3 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.

11.3 Открытая вставка при компиляции С-программ

11.3.3 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.

11.3 Открытая вставка при компиляции С-программ

11.3.3 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.

11.3 Открытая вставка при компиляции С-программ

11.3.3 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.

11.3 Открытая вставка при компиляции С-программ

11.3.3 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;
```

11.3.3 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

      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

11.3 Открытая вставка при компиляции С-программ

11.3.3 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.

◊ Точные оценки выгоды от вставки также очень трудно получить. Если предположить, что

(1) при организации вызова функций превалирующими инструкциями являются инструкции загрузки и освобождения регистров (**`Load`** и **`Store`**), а также инструкции перехода,

и что

(2) эти затраты приблизительно равны для всех вызовов, член, связанный с оценкой выгоды может быть исключен из функции затрат.

11.3 Открытая вставка при компиляции С-программ

11.3.4. 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.

11.3 Открытая вставка при компиляции С-программ

11.3.4. 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. Иногда дублирование заменяют клонированием, чтобы учесть контекст вызова.

11.3 Открытая вставка при компиляции С-программ

11.3.4. 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.

11.3 Открытая вставка при компиляции С-программ

11.3.5. 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).

11.3 Открытая вставка при компиляции С-программ

11.3.5. 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

11.3 Открытая вставка при компиляции С-программ

11.3.6. 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.

11.3 Открытая вставка при компиляции С-программ

11.3.7. Сокращение количества вставок

◊ Решение произвести открытую вставку не только отбирает процедуры, но и определяет порядок, в котором эти процедуры должны быть вставлены.

◊ Пусть $(A \rightarrow B)$ обозначает открытую вставку A в B,

$[X\ Y\ Z]$ – определяет упорядоченную последовательность событий, начинающуюся с X, а

$\{X\ Y\ Z\}$ – три неупорядоченных события.

◊ Точный порядок вставок очень важен: $[(A \rightarrow B)\ (B \rightarrow C)]$ не то же самое, что $[(B \rightarrow C)\ (A \rightarrow B)]$,

так как

$[(A \rightarrow B)\ (B \rightarrow C)] == [(B^* = A \cup B)\ (C^* = B^* \cup C)]$, а

$[(B \rightarrow C)\ (A \rightarrow B)] == [(C' = B \cup C)\ (B' = A \cup B)]$

11.4 Эвристики

11.4.1 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)

11.4 Эвристики

11.4.2 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

11.4 Эвристики

11.4.3 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

11.5 Алгоритм «Агрессивная вставка»

11.5.1 Структура алгоритма

- ◊ Область действия *глобальной* оптимизации ограничена рамками одной процедуры. Открытая вставка – один из способов расширить область действия оптимизирующих преобразований. Ее можно рассматривать как один из видов межпроцедурной оптимизации. После вставки оптимизации, ограниченные пределами процедуры можно применять к объединенному коду вызывающей и вызываемой процедур.
- ◊ Агрессивная вставка включает предварительное клонирование вызываемой процедуры, чтобы учесть в клоне важные особенности контекста вызова.
- ◊ Разница между копией и клоном.

11.5 Алгоритм «Агрессивная вставка»

11.5.1 Структура алгоритма

◊ Термин *агрессивная вставка*, введенный в 1997 году, означает систематическое применение клонирования и вставки процедур, определенных во всех модулях оптимизируемой программы, используя как аннотации пользователя, так и результаты профилирования, с минимальными ограничениями.
Измерения показали, что для набора стандартных тестов обеспечивается ускорение в 1.5 – 2 раза (а не 5 – 10%, достигаемых в остальных оптимизациях).

◊ Агрессивная вставка состоит в выполнении нескольких фаз клонирования и вставки.
Такая многофазная структура выбрана потому, что очень трудно предсказать влияние каждого клона и каждой вставки на оптимизацию. Выполняя все вставки в одну фазу, трудно учесть все особенности вставляемой процедуры прежде всего потому, что они выявляются в процессе первого этапа вставки.

11.5 Алгоритм «Агрессивная вставка»

11.5.1 Структура алгоритма

◊ Управление фазой вставки обеспечивается введением параметра *бюджет*. Бюджет (**B**) это оценка возрастания времени компиляции в связи с увеличением длины вызывающей процедуры в результате данной вставки.

◊ Бюджет вычисляется по формуле

```
B = Q*sizeof(R)
```

По умолчанию для каждой вставляемой процедуры **Q** полагается равным 2, хотя, как показал опыт эксплуатации, среднее значение **Q** = 1.2 (длина процедуры возрастает примерно на 20%).

◊ Пользователь имеет возможность вручную уточнять бюджет в любом направлении.
В начале алгоритма бюджет распределяется между отдельными фазами клонирования и вставки (распределение бюджета тоже может уточняться пользователем).

11.5 Алгоритм «Агрессивная вставка»

11.5.1 Структура алгоритма

◊ Затем начинается *основной цикл*, который состоит в выполнении клонирований и вставок и заканчивается либо при исчерпании бюджета, либо по истечении лимита времени (**`limit`**), отведенного на выполнение данной группы вставок.
Клоны сохраняются в файле (базе данных) **`D`** для ускорения процесса.

◊ В программе (см. следующий слайд) использованы следующие обозначения:

`G = ⟨R, E⟩ -` граф вызовов

`B -` бюджет

`C -` стоимость

`Q -` приращение стоимости

`S[] -` распределение бюджета

`D -` база клонов

`limit`

11.5 Алгоритм «Агрессивная вставка»

11.5.1 Структура алгоритма

```
Inline and Clone(G)
INPUT G:⟨R, E⟩
// оценка стоимости компиляции
   C = 0
   FOREACH R IN G
         C = C + (sizeof(R))
// вычисление бюджета
   growth factor Q = 1.2
   budget B = C * Q
// вычисление распределения бюджета
   S[0] = C + B * 0.2
   . . .
   S[limit-1] = C + B
// клонирование и вставка
   clone database D = { }
   pass number P = 0
   WHILE ( C < B & P < limit ) DO
         C = Clone(G,S[P],C,D)
         C = Inline(G,S[P],C)
         P = P+1
```

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

- Клонирование начинается с **отбора вставляемых процедур**. Вызываемые процедуры исследуются по очереди.
  - Сначала проверяется **допустимость вставки** (например, вставка недопустима, если имеются грубые несоответствия типов между вызывающей и вызываемой процедурами, или нет соответствия между числом передаваемых параметров).
  - Затем проверяется, поставляет ли вызывающая процедура какую-либо интересную дополнительную информацию (например, может в качестве дополнительной информации может передаваться значение параметра по умолчанию, например, 0).
  - Если контекст вызова достаточно интересен, из базы данных запрашивается, как этот контекст используется в вызываемой процедуре. Если выясняется, что вызываемая процедура может быть оптимизирована после замены формальных параметров на фактические, вызов считается подходящим для вставки.

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

- **Отбор вставляемых процедур**.
  - Клонер (фаза клонирования) эффективно совмещает информацию, поставляемую вызывающей процедурой, со своей внутренней информацией для создания спецификации клона.
  - Обычно на этой фазе ограничиваются спецификацией констант, поставляемых вызывающей процедурой. Однако отметим, что на самом деле можно (и нужно) использовать и другую информацию из вызывающей процедуры:
    - сведения об алиасах,
    - сведения о том, что вызываемая процедура не использует часть фактических параметров,
    - сведения о том, что вызывающая процедура не использует возвращаемого значения и т.п.
  - Каждый параметр рассматривается независимо от остальных.

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

- **Формирование групп клонов**.
  - Вычисляется информация о профиле блока, содержащего вызов ("вес" вызова), с целью определить, насколько вызов интересен в аспекте оптимизации.
  - Найдя интересный вызов, Клонер мог бы перейти к построению аналогичных спецификаций для остальных вызовов, но это привело бы к ненужному резкому росту числа клонов.
  - Вместо этого после обнаружения интересного вызова Клонер пытается использовать спецификации клонов для "жадного" построения *группы клонов*: множества вызовов, которые могут безопасно обратиться к клону, удовлетворяющему рассматриваемой спецификации. Для этого исследуется каждый вызов вызываемой процедуры, чтобы убедиться, что контекст вызова совместим со спецификацией, созданной для группы клонов. Если это так, то вызов включается в группу клонов.

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

- **Формирование групп клонов**.
  - После того как группа клонов полностью сформирована, Клонер оценивает выгоду от замены вызываемой процедуры клоном из этой группы.
    При этом учитываются такие факторы как оценка общего количества вызовов, которые будут вызывать клон из группы вместо исходной процедуры, и значение контекстной информации в точке вызова.
  - После исследования вызовов у Клонера будет набор групп клонов, описывающих конкретные клоны, которые могут быть созданы, и оценки выигрыша от создания каждого клона.
  - В заключение Клонер ранжирует группы клонов по выигрышу, "жадно" создавая клоны и модифицируя вызовы, пока не будет исчерпана соответствующая доля бюджета.
  - Любая группа клонов, которая не была обработана на рассматриваемой фазе, отбрасывается.

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

- **Создание клона**.
  - Создать клон очень просто. Копируется промежуточное представление (**IR**) клонируемой процедуры, и каждый формальный параметр, известный из контекста вызова, превращается в локальную переменную вызываемой процедуры и инициализируется подходящей константой, известной из контекста вызова, в блоке **`Entry`** рассматриваемого клона. Если имеются незначительные несоответствия типов, используется явное приведение типов.
  - Клон всегда помещается в тот же модуль, что и клонируемая процедура.
    Если вызывающая процедура, определенная в другом модуле, передает символьную информацию, доступную только в ее модуле (например, адрес статической процедуры), эту информацию необходимо сделать глобальной и присвоить ей уникальное имя, не конфликтующее ни с каким пользовательским именем

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

- **Хранение клонов**.
  - По мере создания клонов каждый клон и связанная с ним спецификация записываются в специальную **базу данных**.
    Эта база данных используется на более поздних фазах: если становится возможной ситуация, когда Клонер выработает такую же спецификацию клона, что и на более ранней фазе, так как промежуточные оптимизации добавили информацию к вызовам, которые до этого не заслуживали внимания.
    В этом случае наличие клона в базе данных приводит к его повторному использованию, ускоряя работу Клонера.

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

- **Модификация вызовов**.
  - Модификация вызовов группы клонов, чтобы обеспечить вызов требуемого клона, тоже очень проста.
    Спецификация клона описывает сигнатуру новой процедуры, следовательно, любые параметры, включенные в клон могут редактироваться согласно списку фактических параметров. После этого вызов модифицируется, чтобы ссылаться на клон, а не на исходную процедуру.
  - Такая модификация, естественно влечет соответствующие изменения в графе вызовов, чтобы тот мог отражать новые отношения между вызывающей, вызываемой и клонируемой процедурами и клоном.
    В частности, если все вызовы клонируемой процедуры будут заменены на вызовы одного из ее клонов, клонируемая процедура может стать недостижимой в графе вызовов и будет удалена.

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

- **Модификация вызовов**.
  - Оценивая влияние конкретной группы клонов на время компиляции, Клонер пытается предсказать последующее удаление клонируемой процедуры, так как считается, что группа клонов, для которой клонируемая процедура будет гарантированно удалена, не влияет на время компиляции.

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

```
Clone(G,B,C,D): returns C
    INPUT
      call graph G:⟨R, E⟩
      budget B
      current cost C
      clone database D
    ALGORITHM
      //Настройка: создать дескрипторы параметров
      //и дескрипторы контекстов вызова
      FOR EACH routine R IN G
        create parameter-usage descriptor P(R)
        FOR EACH edge E IN G
          create calling-context descriptor S(E)
```

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

```
// Построение групп клонов; R – вызываемые
// процедуры intersect() и matches() проверяют
// совместимость со спецификацией
     FOREACH edge E in G
     R = E.target
     IF(clonable(R)& clonable(E))THEN
// CS – спецификация клона
// CG – группа клонов
          CS = intersect(S(E),P(R))
       IF( CS is  nonempty)THEN
         CG = (R,CS,E)
         FOREACH edge EJ incident on R
         IF(clonable(E')& matches(S(E),CS))THEN
           add E’to CG
           estimate benefit of CG
```

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

```
// отбор клонов
     sort CGs by benefit; C' = C
     FOREACH CG IN CGs
       cost x = (sizeof(R))
       IF ( C' + X < B ) THEN
         accept CG; C' = C' + X
// создание клонов и фиксация вызовов
     FOREACH CG IN accepted CGe
       IF(!lookup(D, R, CS> > THEN
         R' =  make  clone (R, CS)
         add database entry ( R, CS, R′ )
       FOREACH edge E' IN CC
         change target of E' from R to R'
```

11.5 Алгоритм «Агрессивная вставка»

11.5.2 Алгоритм. Фаза клонирования

```
//  оптимизация клонов и перестройка ГВ
     FOREACH newly created clone R'
      optimize(R′)
      C = C + (sizeof(R'))
```

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

- **Общая структура фазы вставки** аналогична структуре фазы клонирования.
- **Отбор процедур и клонов для вставки**
  - Вставщик (**`Inliner`**) рассматривает все вызовы, выявляя любые **ограничения** на вставку (языковые, технические, прагматические, пользовательские). Выявляются также **неверные вызовы**, содержащие ошибки (несогласованность типов, несогласованность по местности операций и т.п.)
  - Технические ограничения на вызовы – это случаи, когда информация, связанная с вызываемой процедурой противоречит информации, связанной с вызывающей процедурой. Например, в коде вызывающей процедуры может быть указано, что перестановка операций с плавающей точкой разрешена, тогда как вызываемая процедура может декларировать, что такая перестановка неприемлема. По большому счету ограничения такого рода призваны упростить задачу представления информации.

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

- **Отбор процедур и клонов для вставки**
  - К прагматическим проблемам относятся такие вопросы как обработка вызываемых процедур, использующих **`alloca()`** для динамического расширения стека, или вставка вызова, фактические параметры которого описывают перекрывающиеся области памяти, а вызываемая процедура разработана в предположении, что ее формальные параметры не могут быть алиасами.
  - Пользовательские ограничения получаются в результате использования различных опций командной строки и прагм.

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

- **Ранжирование отобранных вставок.**
  - После того, как требуемые вызовы, заменяемые вставкой, отобраны, им на время выполнения алгоритма присваивается показатель качества (приоритет). Высший приоритет присваивается наиболее частым вызовам. Вызовам, находящимся в блоках, выполняемых менее часто, начисляются штрафы. Это помогает избегать вставок в не критический путь; невыполнение этого требования может вызвать повышение регистрового давления, приводящего к сливам на критических путях, что существенно снижает производительность.
  - После этого Вставщик следует по списку вставок в порядке их приоритетов. Оценивается влияние каждой вставки на время компиляции и, в пределах текущего бюджета, вставка принимается к исполнению.

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

- **Взаимодействие между вставками.**
  - Вычисление влияния на время компиляции осложняется взаимодействием между вставками.
    Например, если **A** вызывает **B** и **B** вызывает **C**, цена вставки **B** в **A** зависит от того, была ли предварительно выполнена вставка **C** в **B**.
  - Для моделирования указанной зависимости, Вставщик поддерживает график (расписание) порядка, в котором он будет выполнять принятые вставки.
    В общем и целом Вставщик пытается обрабатывать граф вызовов снизу-вверх.
  - Чтобы вычислить цену вставки **B** в **A**, дескриптор указанной вставки вставляется в соответствующую точку графика вставок. Если потом выясняется, что **B** is является целью более ранней вставки (или вставок), то для вычисления цены оптимизации **A** используется оценка размера **B** после этих вставок.

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

◊ **Вставщик** "жадно" принимает и обрабатывает вызовы пока не исчерпается выделенная доля бюджета. При этом оставшиеся жизнеспособные вызовы отбрасываются, так как не осталось времени для их обработки.

◊ В случае косвенных вызовов (вызовов по указателю) вызываемая функция становится известной во время выполнения, так что эти вызовы не могут быть вставлены или клонированы. Однако, иногда удается во время компиляции выявить вызываемую функцию.
Например, Вставщик может клонировать вызовы, в которых вызывающая процедура передает указатель вызываемой процедуре, которая использует для косвенного вызова формальный параметр. Последующая оптимизация "Распространение констант" может вычислить и передать информацию, необходимую для превращения косвенного вызова в прямой, что позволит на более поздней фазе клонировать или вставить этот вызов.

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

```
Inline(G,B,C): returns C
INPUT
 call graph G:(R,E)
 budget B
 current cost C
ALGORITHM
 // screen inline candidates
 FOR EACH edge E IN G
   IF(inlineable(E))  > THEN
   accept E; compute benefit(E)
```

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

```
// отобрать вызовы для вставки
// отсортировать отобранные вызовы E'
// по выгоде
 C' = C
 FOR EACH accepted edge E
   insert  E into  schedule
   cost X =
   (sizeof(E.target+E.source)²
   - (sizeof(E.target)²
   C″ = C'
   C' = C' + X
 IF ( E.target is source in later inline )
 THEN adjust C' for cascaded cost
 IF ( C' >  B  ) THEN
   remove  E  from  schedule
 C′= C″
```

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

```
// выполнить вставки
 FOR EACH scheduled edge E
   inline E.target into  E.source
// оптимизировать вставки и перестроить ГВ
 FOR EACH routine inlined into R'
   optimize(R’)
   C = C + (sizeof(R’))
```

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

- **Обоснование эвристик**.
  - Методики, позволяющей определять во время компиляции оптимальное множество вставок или клонов, к сожалению, не существует. Необходимо оценивать как выигрыш времени во время исполнения программы, так и затраты на оптимизацию во время компиляции. Более того, при разработке любой программы разумных размеров применяется и клонирование, и открытая вставка. При этом даже упрощенный вариант проблемы эквивалентен известной задаче о рюкзаке, которая, как известно, является NP полной.
  - Каждый вызов может быть либо заменен вставкой, либо клонирован, причем окончательный код существенно зависит от порядка клонирований и вставок. Поэтому порядок вставок и клонирований должен определяться с помощью эвристики. Для выбранной эвристики полезно показать, что решения, принимаемые с ее помощью, выглядят разумными.

11.5 Алгоритм «Агрессивная вставка»

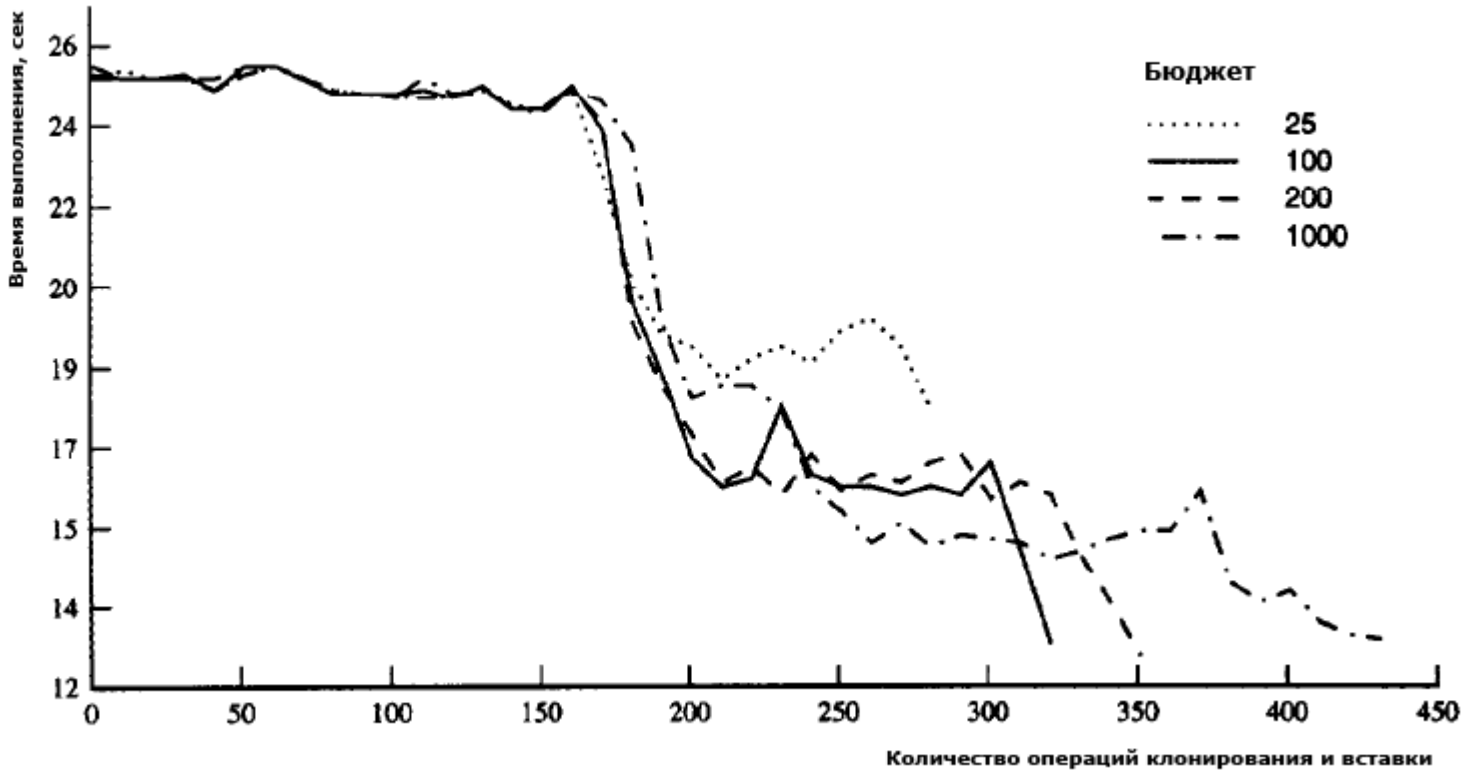
11.5.3 Алгоритм. Фаза вставки

- **Обоснование эвристик**.
  - Для оценки качества рассмотренных эвристик были проведены экспериментальны исследования на тестах (бенчмарках) из *SpecInt*.
    Результаты для теста **`022.li`** в виде графика, иллюстрирующие одну из методик оценки качества эвристик см. на слайде 29.
  - Предоставляемый бюджет менялся от сравнительно малого значения 25 до очень большого бюджета 1000. Для каждого уровня бюджета несколько раз выполнялась компиляция теста. Во время каждой компиляции Вставщик искусственно останавливался после определенного количества вставок и/или клонирований.
  - Полученные кривые показывают дополнительную пользу каждой последующей вставки или клонирования.
    Как можно видеть из графика, лишь очень немногие вставки или клоны оказывают неблагоприятное воздействие на производительность.

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

◊ **Обоснование эвристик**



11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

- **Обоснование эвристик**.
  - Как показали экспериментальные исследования, после того как выделенный бюджет достигает достаточно большого значения (например, 100), дальнейшее приращение производительности за счет дополнительных вставок **для большей части тестов** из *SpecInt* прекращается (производительность достигает своего предельного значения). Поэтому в подсистеме компилятора, реализующей агрессивную вставку, значение бюджета, назначаемое по умолчанию, было выбрано равным 100, что позволило максимизировать производительность тестов из *SpecInt*, без выполнения ненужных вставок

11.5 Алгоритм «Агрессивная вставка»

11.5.3 Алгоритм. Фаза вставки

- **Обоснование эвристик**.
  - Как показало исследование вызовов в тестах (бенчмарках) из *SpecInt*, среди них имеется значительное количество межмодульных вызовов.
    Обеспечение возможности заменять вставками межмодульные вызовы имеет решающее значение для обеспечения хорошей производительности оптимизируемой программы.