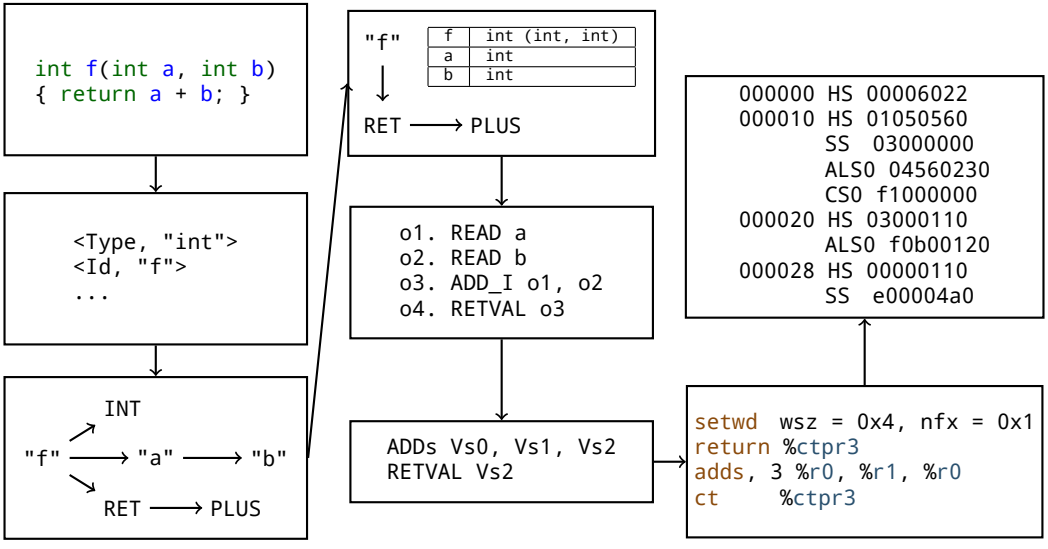
Проектирование компиляторов

Лекция 10.
Оптимизации потока управления.

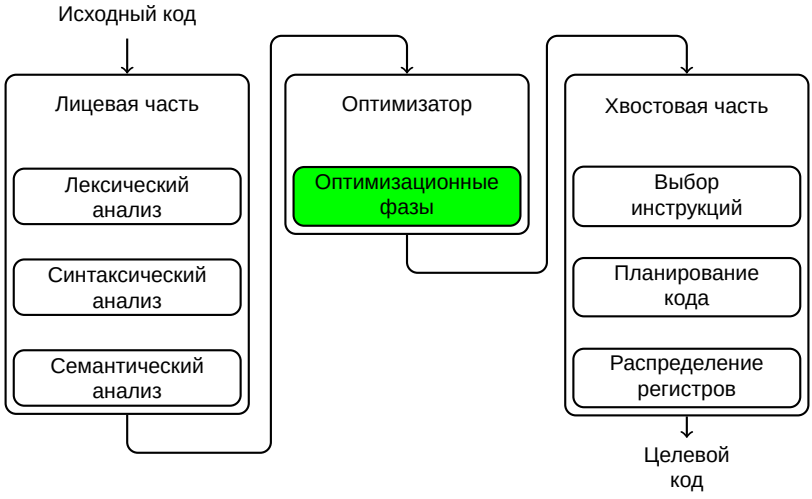
Маркин А. Л.
alexanius@gmail.com

2021

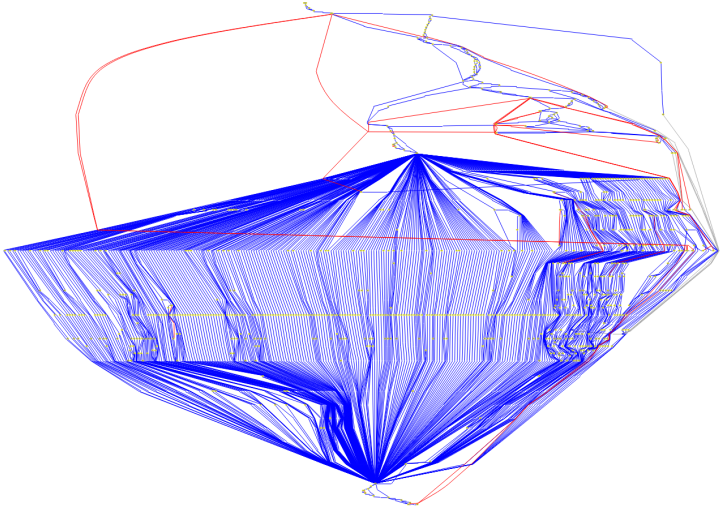
Компиляция программы



Компиляция программы



Граф потока управления



Граф потока управления

Граф потока управления состоит из линейных участков и возможных переходов между ними. Граф обладает стартовым узлом, доминирующим над всеми его узлами и стоповым узлом, постдоминирующем все узлы графа.

Линейный участок (basic block) — последовательность операций максимальной длины от входа в участок до операции перехода.

Оптимизации потока управления

Переход (передача управления) — операция, меняющая адрес текущей инструкции (и контекст в случае вызова функции). Каждый переход несёт в себе накладные расходы на подготовку, а подкачка инструкций затруднена т.к. направление перехода заранее неизвестно.

Размер линейного участка влияет на возможность обнаружения параллелизма инструкций и качество планирования.

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

Устранение избыточности

Проблема

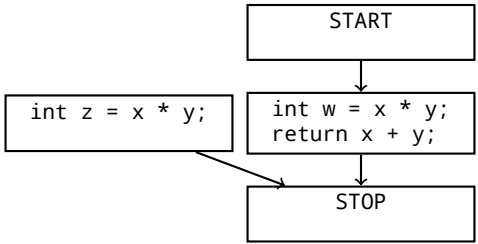
Код называется **мёртвым**, если он исполняется, но его результат не используется.

Код называется **недостижимым** если до него не доходит управление независимо от входных данных.

```
int f(int x, int y)
{
    int w = x * y; // Мёртывый код
    return  x + y;
    int z = x * y; // Недостижимый код
}
```

Удаление недостижимого кода

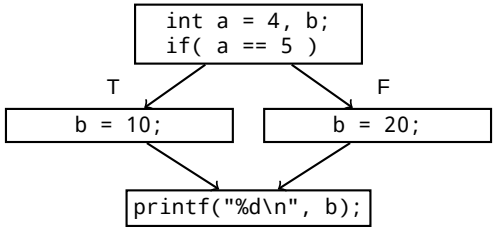
Недостижимый узел не имеет пути от стартового узла процедуры:



Удаление недостижимого кода

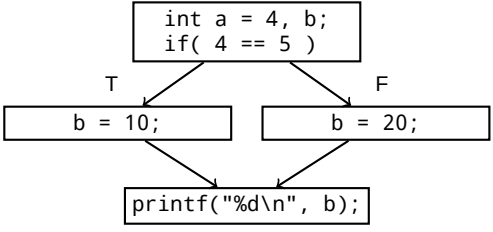
Недостижимые узлы могут получиться как из исходного кода, так и в следствии оптимизаций:

```
int a = 4, b;

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

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

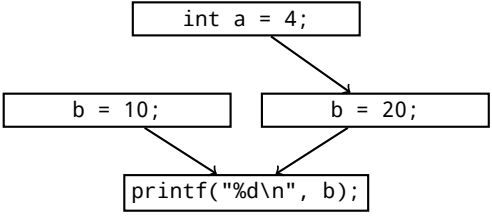


Удаление недостижимого кода

После распространения констант:

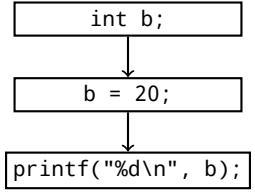


После удаления мёртвого кода:



Удаление недостижимого кода

После удаления недостижимого кода:



Удаление недостижимого кода

Преимущества:

- Упрощение промежуточного представления
- Уменьшение размера итоговой программы
  - Как побочный эффект, возможно ускорение за счёт оптимизации кэша данных

Упрощение переходов

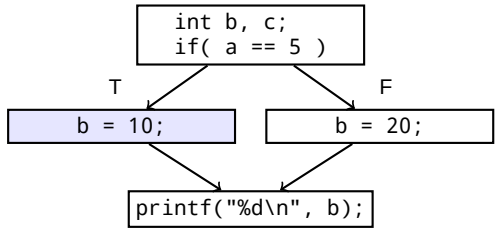
Для уменьшения количества переходов и увеличения размера линейных участков существуют различные методы оптимизации графа потока управления:

- Удаление пустых блоков условных конструкций
- Упрощение конструкций со статически вычислимыми условиями
- Упрощение условий, основанных на подвыражениях

Упрощение переходов

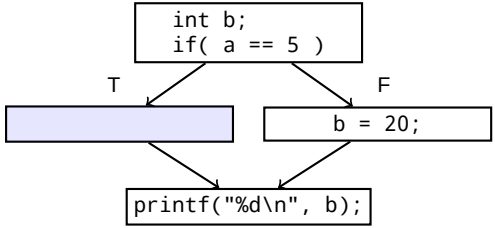
Откуда берутся пустые блоки:

```
void foo(int a)
{
    int b, c;

    if( a == 5 )
        c = 10; // Мёртвый код
    else
        b = 20;

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



Упрощение переходов

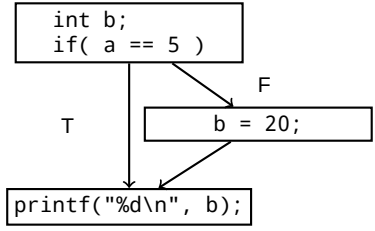
После удаления мёртвого кода:



Упрощение переходов

Даже если линейный участок не содержит кода, переход всё равно будет представлен соответствующей инструкцией, исполнение которой занимает время и может вести к другим накладным расходам.

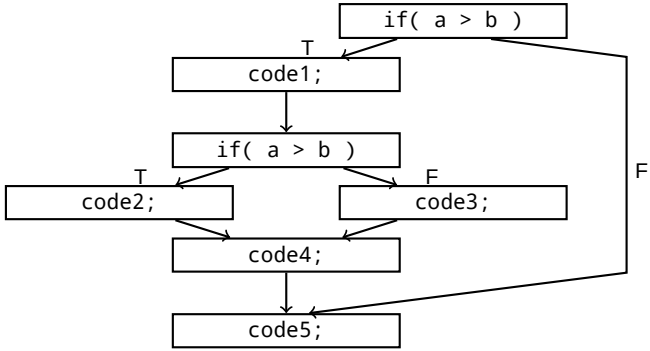
Переход на пустой линейный участок является избыточным, и его можно удалить, не изменив поведение программы:



Устранение ветвлений

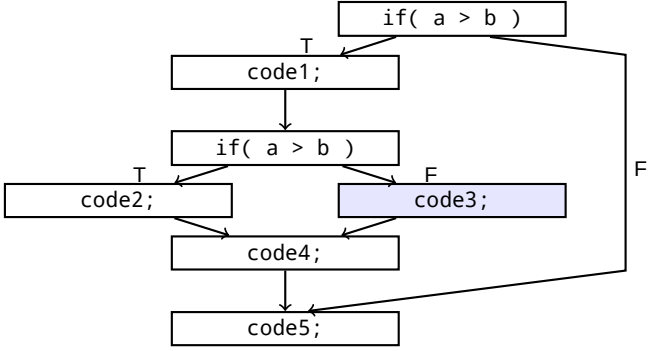
Проблема

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



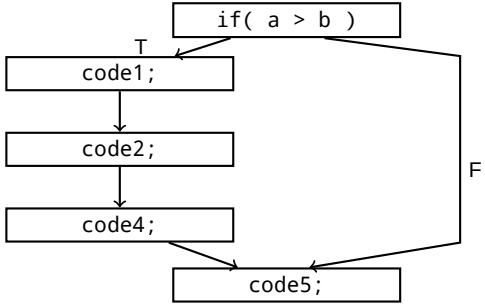
Упрощение ветвлений

Если переменные a и b не изменяются между двумя условиями, то блок кода с `code3;` никогда не не исполнится:



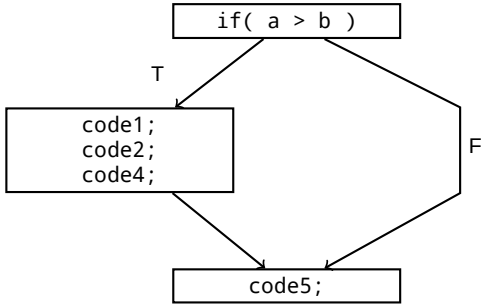
Упрощение ветвлений

После упрощения граф потока управления выглядит следующим образом:



Упрощение ветвлений

Если узел N_1 имеет только одну безусловную исходящую дугу в N_2, а узел N_2 имеет только одну входящую дугу из N_1, то эти узлы можно объединить в один:



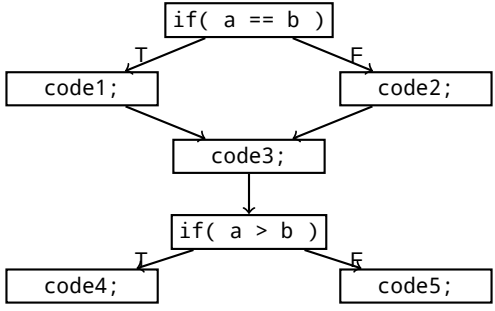
Упрощение ветвлений

Преимущества:

- Уменьшение количества исполняемых операций
- Уменьшение размера программы
- Создание контекста для других оптимизаций

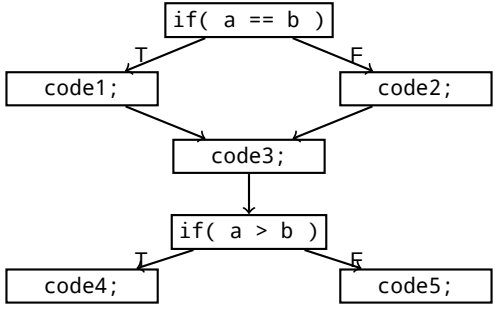
Проблема

Гораздо чаще условия не являются дублирующими:



Проблема

Гораздо чаще условия не являются дублирующими:

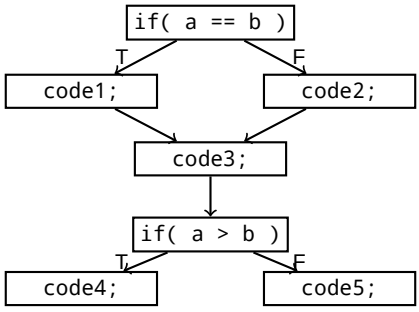


Можно заметить что при выполнении первого условия результат второго будет заранее известен (при условии неизменности переменных a и b).

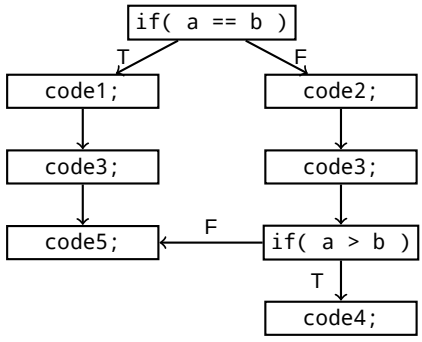
Упрощение ветвлений

При истинном значении первого условия возможно не выполнять вторую проверку, но для этого следует преобразовать граф потока управления:

До преобразования



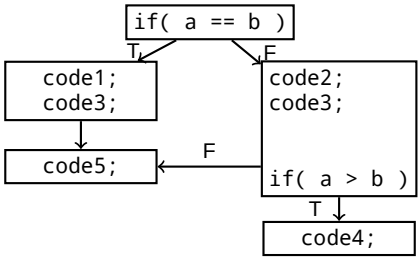
После преобразования



Преобразование создаёт копию узла, содержащего `code3`.

Упрощение ветвлений

Далее произойдёт объединение линейных участков, что создаст контекст для других оптимизаций:



Преимущества:

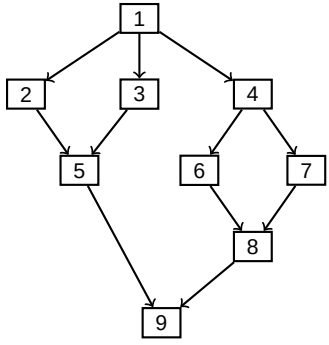
- Уменьшение количества исполняемых операций
- Уменьшение количества переходов
- Создание контекста для других операций

Недостатки:

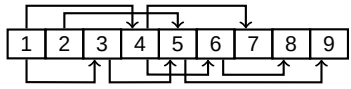
- Увеличение размера процедуры

Проблема

В исполняемом файле линейные участки размещаются последовательно



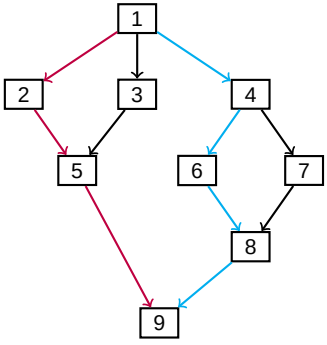
Расположение линейных участков в исполняемом файле:

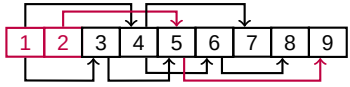


Для перехода между соседними линейными участками (например для перехода $1 \to 2$) не нужно выполнять инструкцию перехода. Такие переходы называются **«провалы»** (fall).

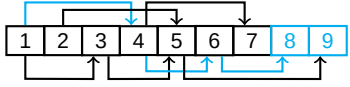
Проблема

В зависимости от маршрута будет выполнено разное количество переходов:





Один «провал»
Два перехода

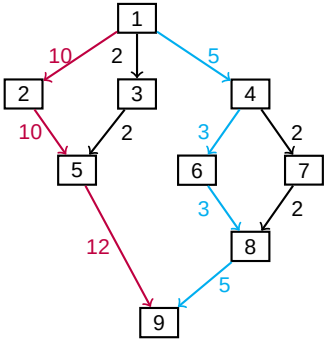


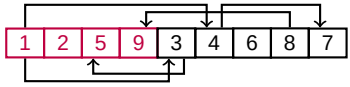
Один «провал»
Три перехода

Чем больше выполняется переходов тем медленнее работает приложение.

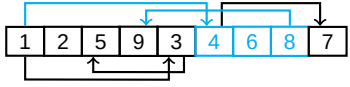
Размещение линейных участков

Если известен профиль исполнения, то линейные участки можно расположить оптимальным образом:





Четыре «провала»
Ноль переходов



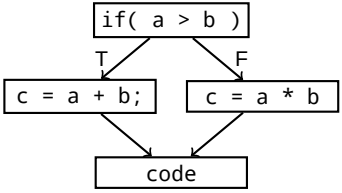
Два «провала»
Два перехода

Оптимальное расположение линейных участков устраняет лишние переходы и улучшает работу с кешем инструкций.

Аппаратные возможности устранения ветвлений

Проблема

Бывают случаи когда все возможности для упрощения ветвлений уже исчерпаны:




```
{
 a > b -> %p1
}
{
 ct false | %p1 == f
}
{
 c = a + b
 ct tail
}
false:
{
 c = a + b
}
tail:
{
 code
}
```

Предикатное исполнение

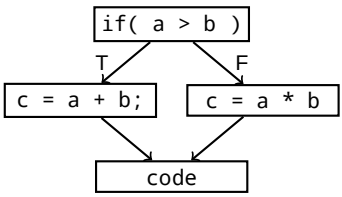
У процессоров с широкой командой существует аппаратный способ устранения ветвлений — **предикатное исполнение**.

Предикат — значение в специальном регистре, которое разрешает или запрещает процессору исполнять заданную инструкцию.

Процессор Эльбрус содержит 32 предикатных регистра. Под предикат может быть поставлена любая операция. В процессорах Sparc v9 существует один предикат, который может быть использован для команды `MOV`.

Предикатное исполнение

Предикатное исполнение позволяет устранять переходы и объединять ветвления в один линейный участок:




```
{
 a > b -> %p1
}
{
 ct false | %p1 == f
}
{
 c = a + b
 ct tail
}
false:
{
 c = a + b
}
tail:
{
 code
}
```

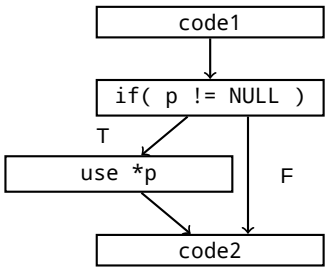
```
{
  %p = a > b
}
{
  c  = a + b | %p == true
  c  = a * b | %p == false
  code
}
```

Предикатное исполнение

Предикатное исполнение даёт возможность сливать более сложные ветвления:

Проблема

Иногда зависимости по управлению мешают поднимать вверх тяжёлые операции:



Операция загрузки значения по адресу, хранящемуся в p может занимать длительное время, в которое процессор будет простаивать.

Проверка в узле-предшественнике не позволяет поднять операцию загрузки выше.

Разыменование нулевого адреса вызывает прерывание.

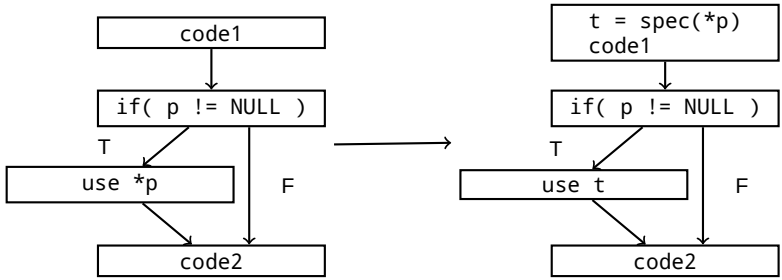
Спекулятивное исполнение

У процессоров на основе широкой команды существует аппаратный способ исполнения инструкций, способных вызвать прерывание.

Спекулятивный режим исполнения (режим отложенных прерываний) позволяет выполнить инструкцию, но в случае возникновения прерывания отложить реакцию на него.

Прерывание будет вызвано когда неспекулятивная операция попытается использовать результат операции, вызвавшей прерывание.

Спекулятивное исполнение



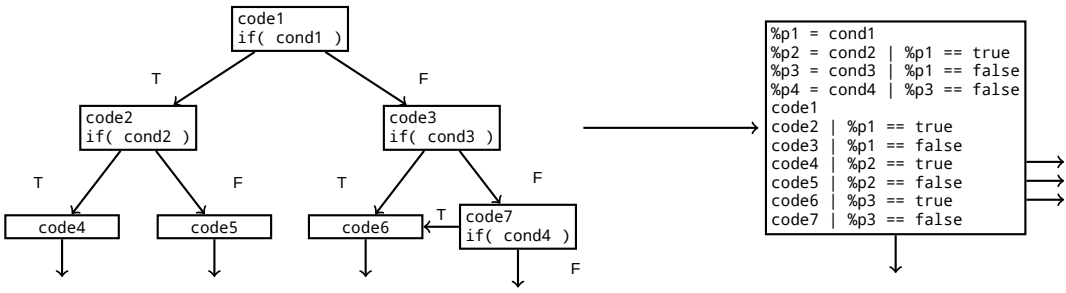
Загрузка значения из p начинается за долго до сравнения. Если он окажется не нулевым, то в моменте использования не потребуется ожидать значения загрузки.
Если p окажется нулевым, то мы не будем использовать результат спекулятивной операции, и отложенное прерывание никогда не вызовется.

Слияние ветвлений

Слияние ветвлений (регионирование, if-conversion) — объединение нескольких линейных участков в один.

Оптимизация выбирает множество узлов (регион) на ациклическом участке. Это множество имеет только один входной узел, выходов может быть несколько. Далее производится слияние узлов региона в один узел при помощи предикатного режима исполнения.

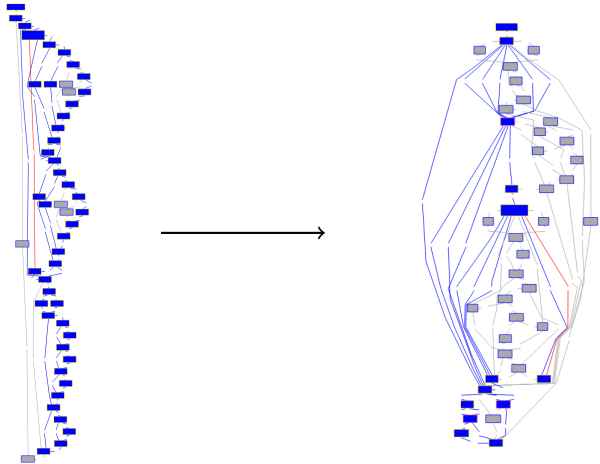
Слияние ветвлений



Слияние ветвлений приводит к тому что узел графа управление имеет несколько операций выхода не в последней операции. Строго говоря, такой узел не является линейным участком, и называется **гиперузел**.

Слияние ветвлений

Пример работы слияния ветвлений на реальной процедуре:



Слияние ветвлений

Преимущества:

- Уменьшение количества переходов
- Увеличение размера линейных участков
- Хороший контекст для последующих оптимизаций и наращивания параллелизма

Недостатки:

- Увеличение размера процедуры (алгоритм слияния ветвлений агрессивно дублирует узлы)
- Предикатное представление кода сложно обрабатывать

Литература

- *S. S. Muchnick* Advanced Compiler Design and Implementation.
- *K. D. Cooper, L. Torczon* Engineering a compiler.
- *R. Allen, K. Kennedy* Optimizing Compilers for Modern Architectures: A Dependence-based Approach.
- *А. В. Ахо, М. С. Лам, Р. Сети, Д. Д. Ульман* Компиляторы: принципы, технологии и инструменты.
- *А. Ю. Дроздов, С. В. Новиков* Исследование методов преобразования программы в предикатную форму для архитектур с явно выраженной параллельностью
- *В. Ю. Волконский, А. В. Брегер, А. Ю. Бучнев, А. В. Грабежной, А. В. Ермолицкий, Л. Е. Муханов, М. И. Нейман-заде, П.А. Степанов, О.А. Четверина* Методы распараллеливания программ в оптимизирующем компиляторе