

Динамические оптимизации в LLVM

Докладчик: Меркулов Алексей, МГУ ВМК, 3 курс,
кафедра системного программирования

Поведение программы

Пусть Вася и Петя любят играть
в какую-нибудь стрелялку

Вася:

Любит карты с жуткими закрытыми помещениями
Играя часто со страху сохраняется через 10 секунд
Процессора у него - Athlon

Петя:

Любит открытые пространства
Практически не сохраняется
Процессора у него — Core 2 Duo

Если посмотреть соотношение времени,
затрачиваемого на выполнении соответствующих инструкций
движка игры, то оно получится разным!

Цель

Создать систему, способную на этапе исполнения программы находить «горячие» места, оптимизировать их, перекомпилировать и вставить в работающую программу «на лету».

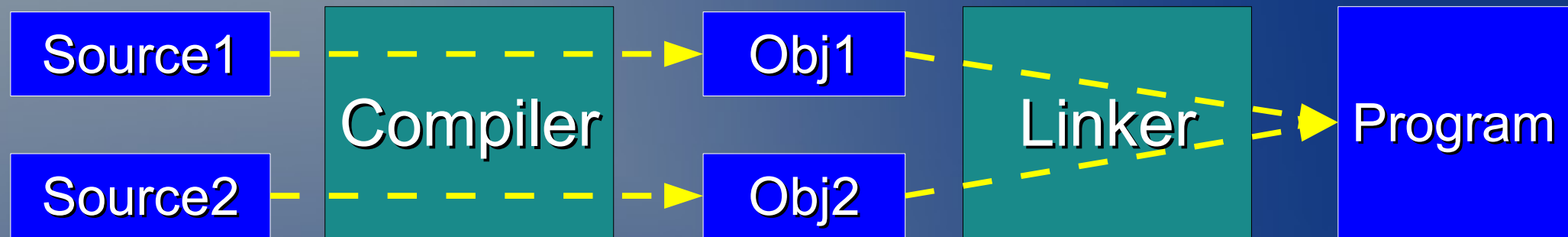
Задачи

Разработать драйвер на основе LLVM, который будет принимать решения о перекомпиляции

Разработать систему сбора информации (профиля) о программе

Добавить в LLVM профиле-зависимые оптимизации или параметризовать существующие профилем.

Классическая модель компиляции



Например

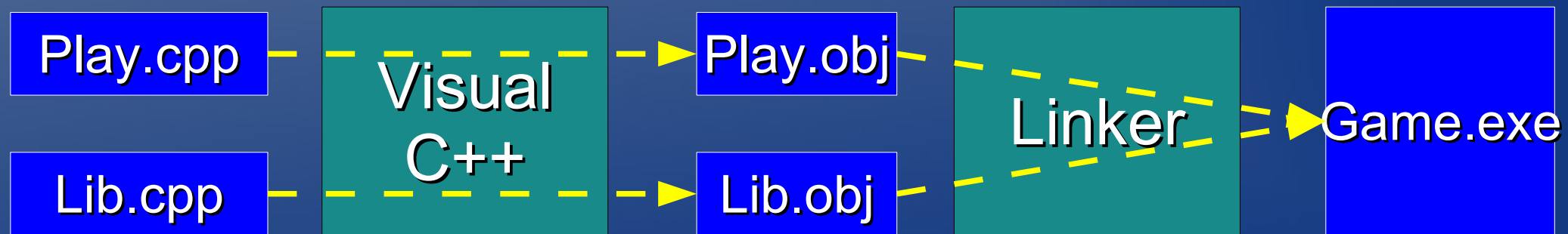
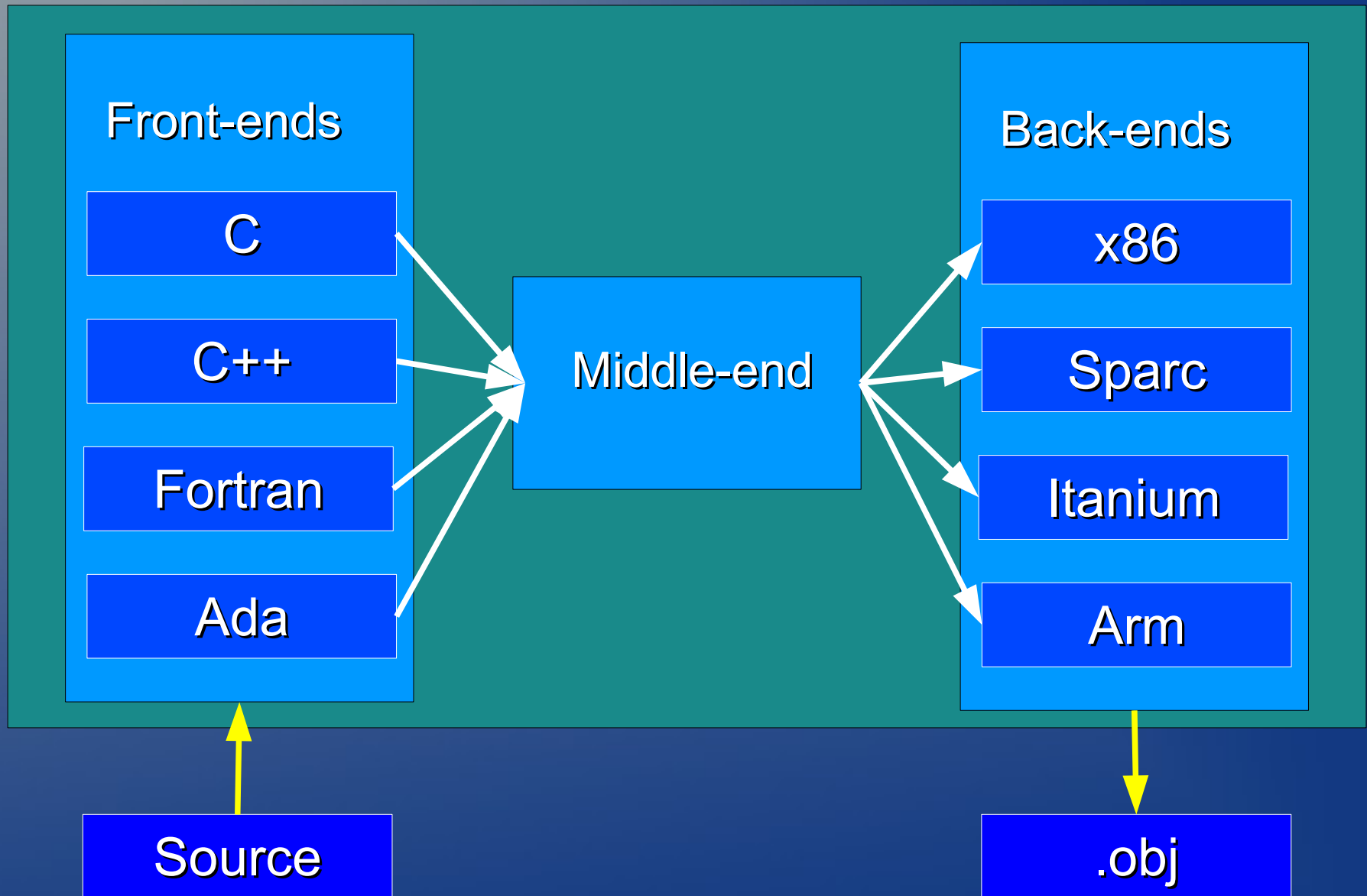
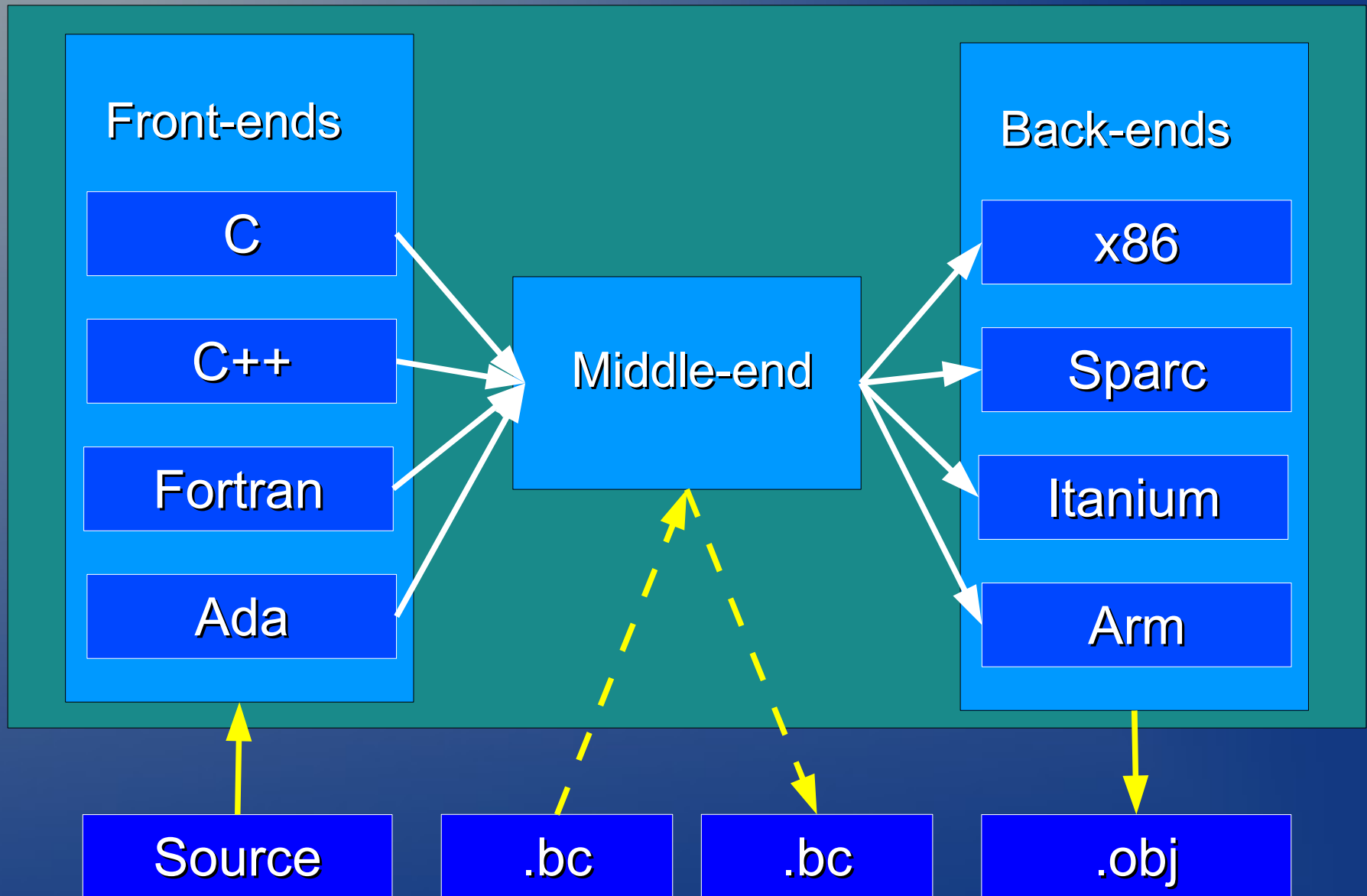


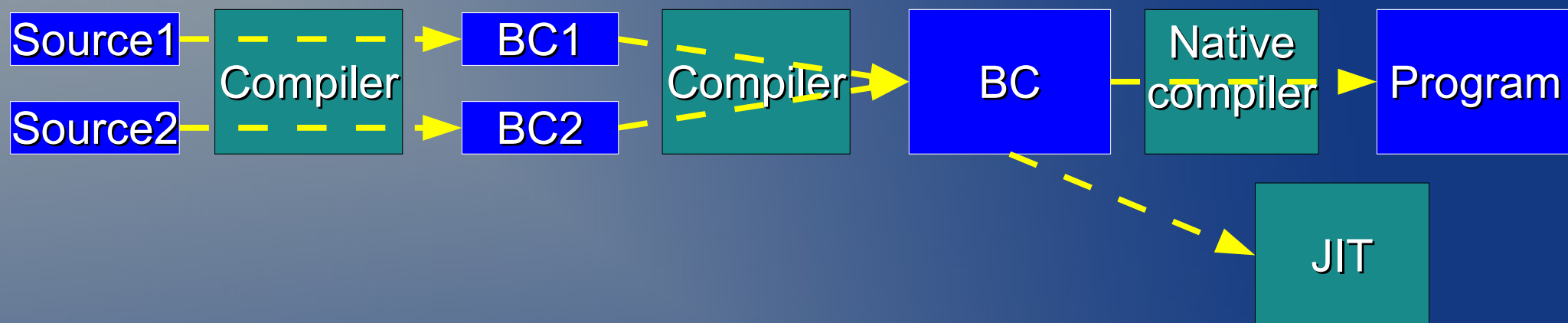
Схема типичного компилятора



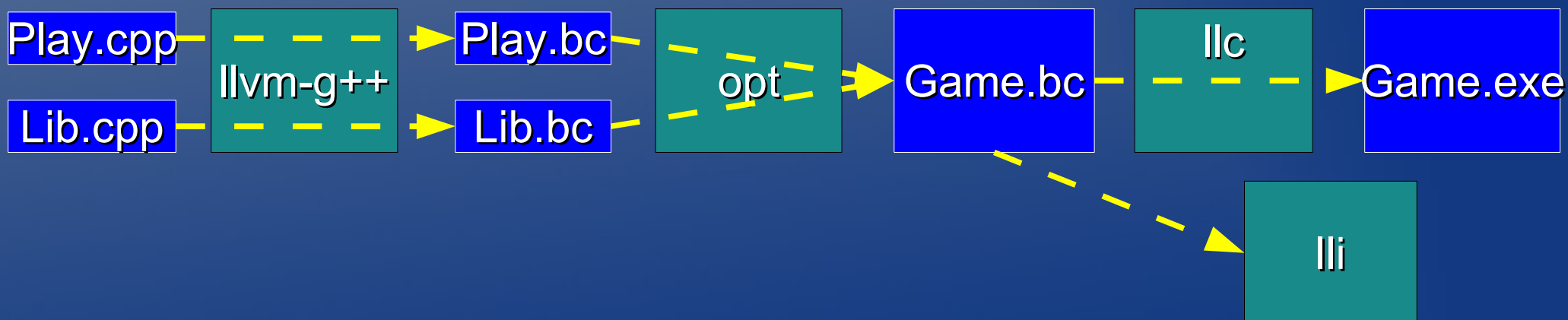
LLVM — Low Level Virtual Machine



LLVM модель компиляции



Например



Ассемблер

```
int x = 0, y = 0;
```

```
scanf("%d %d", &x, &y);
```

```
if (x > y)
```

```
    printf ("more!");
```

```
else
```

```
    printf ("less!");
```

```
return 0;
```

```
xor eax,eax  
mov dword ptr [esp],eax  
mov dword ptr [esp+4],eax
```

```
lea  eax,[esp+4]  
push eax  
lea  ecx,[esp+4]  
push ecx  
push offset string "%d %d"  
call scanf
```

```
mov edx,dword ptr [esp+0Ch]  
add esp,0Ch  
cmp edx,dword ptr [esp+4]  
jle else_label
```

```
push offset string "more!"  
call printf  
add  esp,4  
jmp  ret_label
```

```
else_label:  
push offset string "less!"  
call printf  
add  esp,4
```

```
ret_label:  
xor eax,eax  
add esp,8  
ret
```

Базовые
блоки

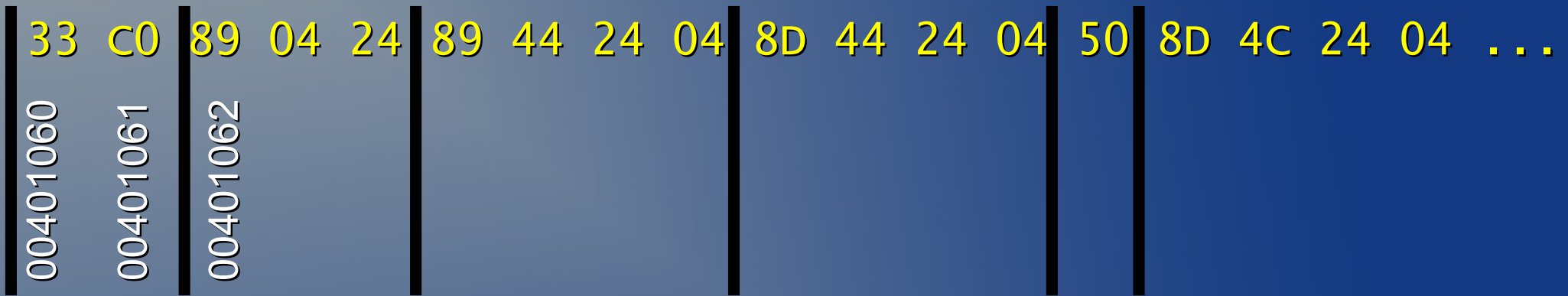
A

B

C

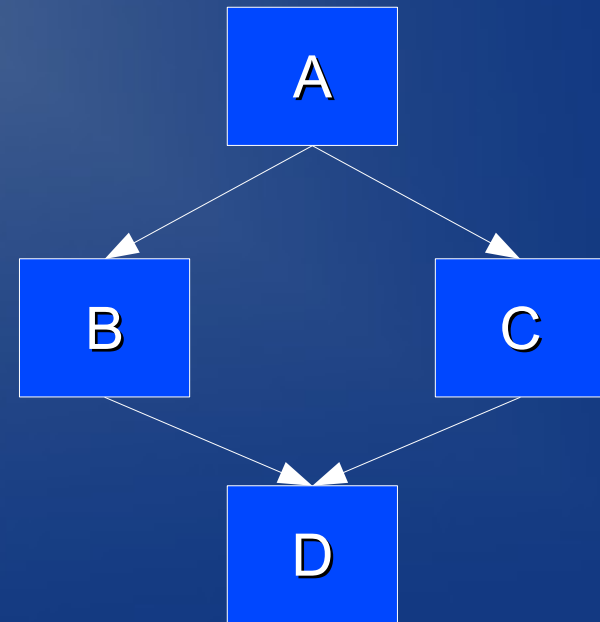
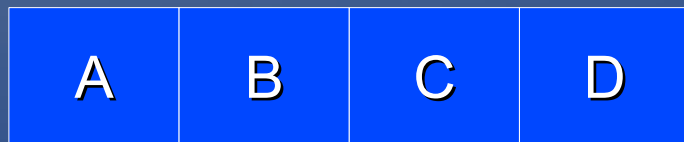
D

Машинный код



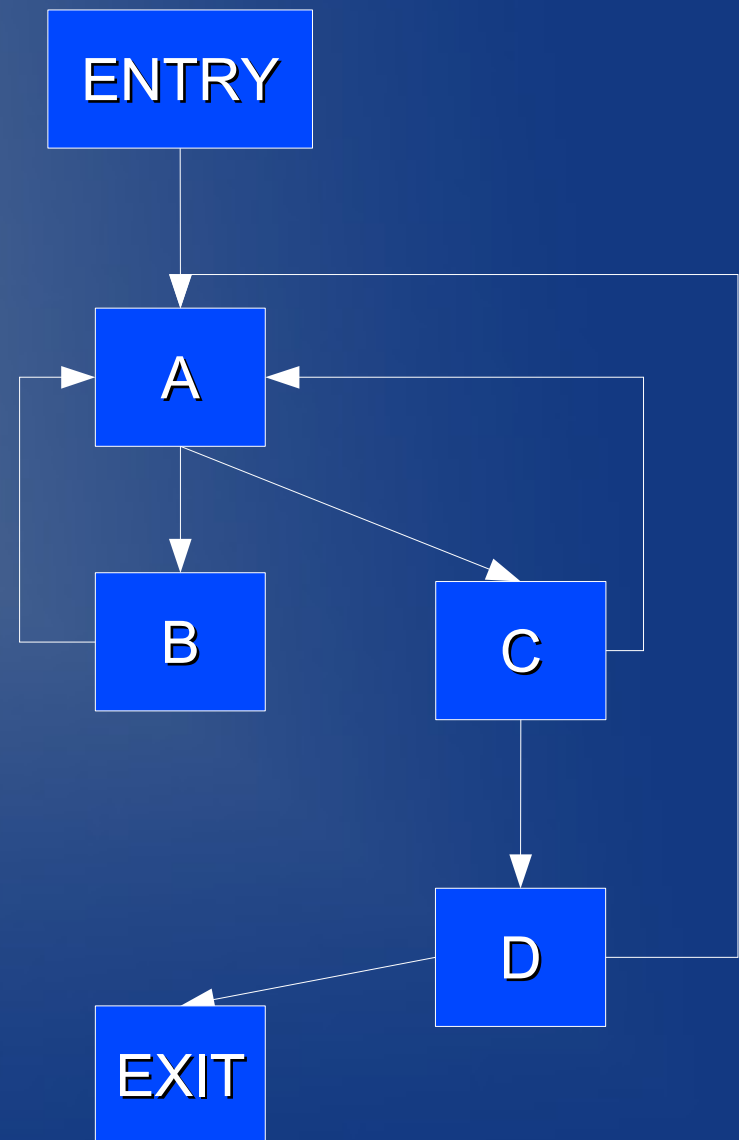
Логически же они представляют такую структуру

Базовые блоки идут в памяти друг за другом



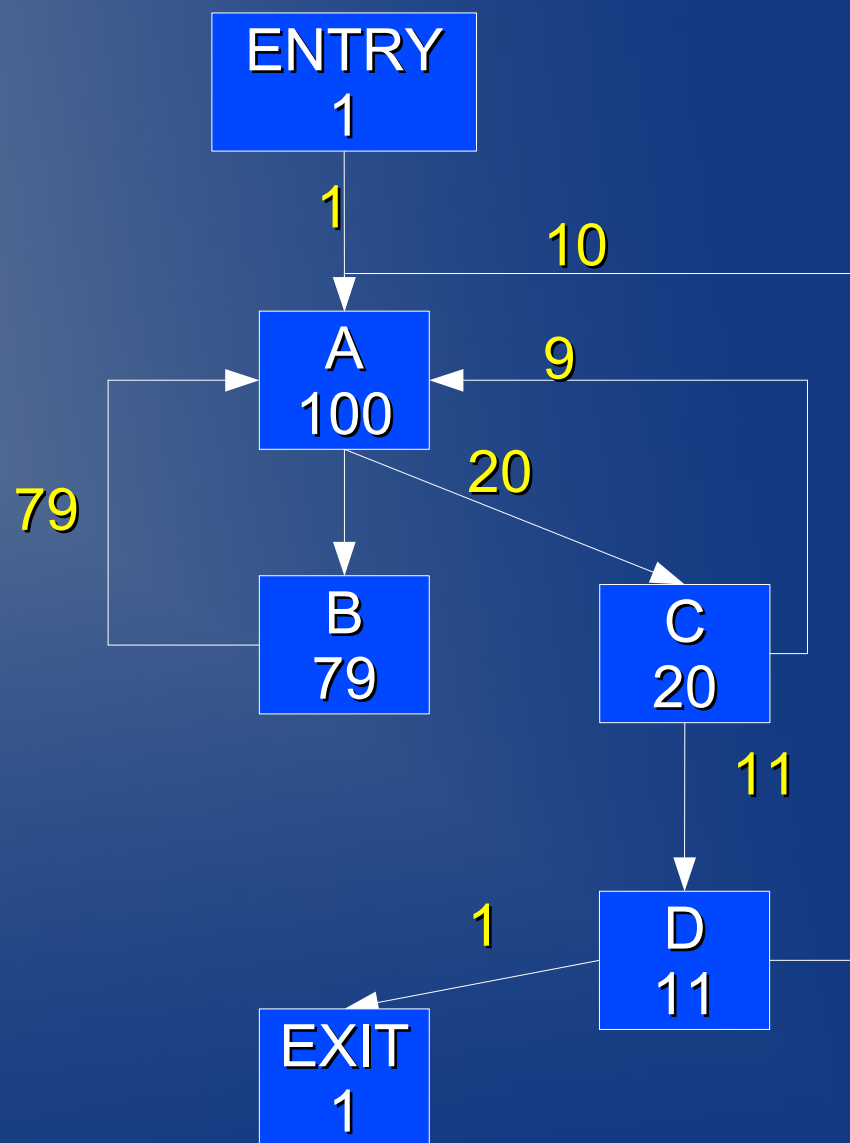
Граф потока управления Control flow graph (CFG)

```
while (1)
{
    A; if (cmp1)
        B;
    else
    {
        C; if (cmp2)
        {
            D; if (cmp3)
                return;
        }
    }
}
```

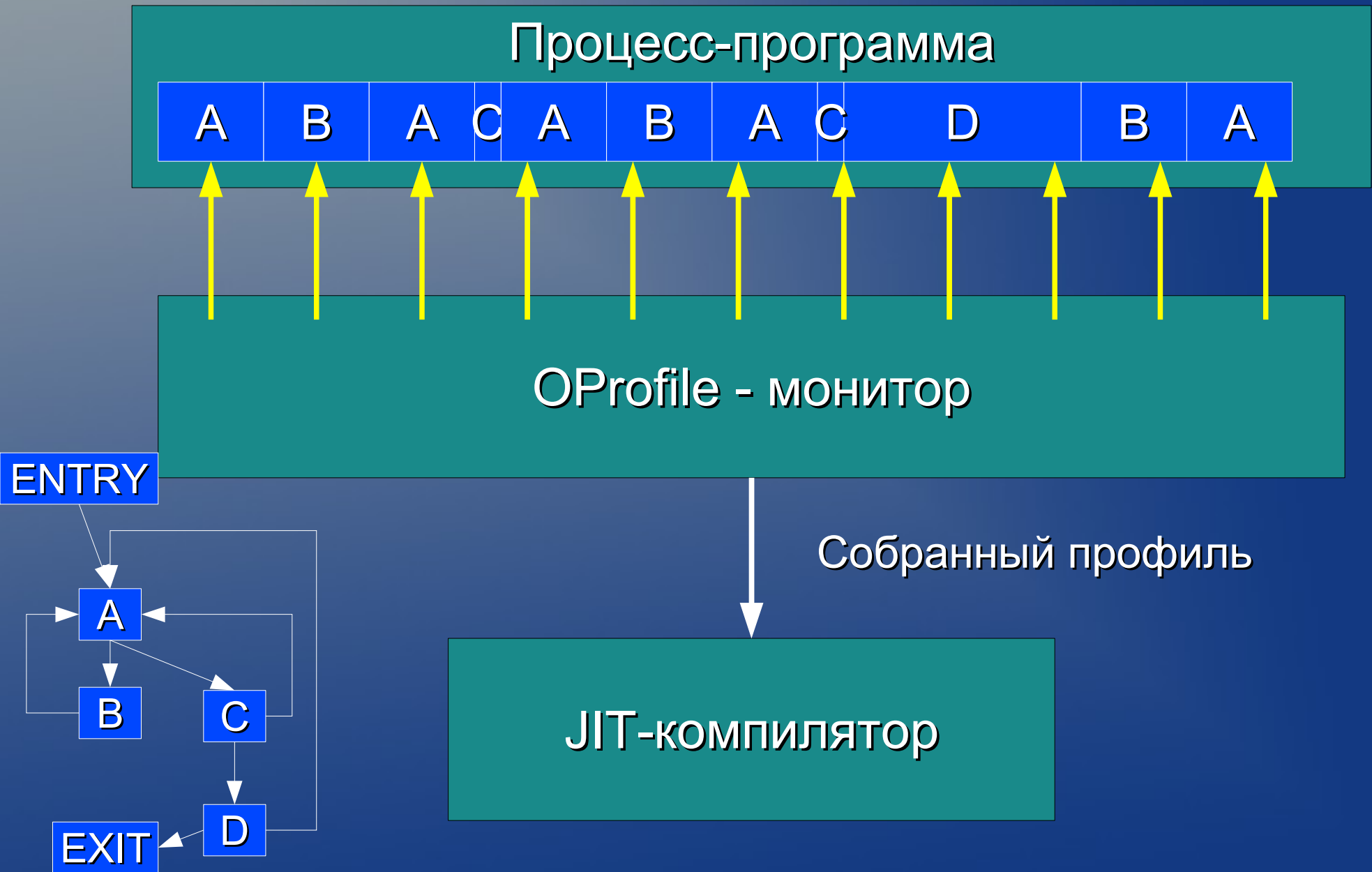


Профиль программы

```
while (1)
{
    A; if (cmp1)
        B;
    else
    {
        C; if (cmp2)
        {
            D; if (cmp3)
                return;
        }
    }
}
```



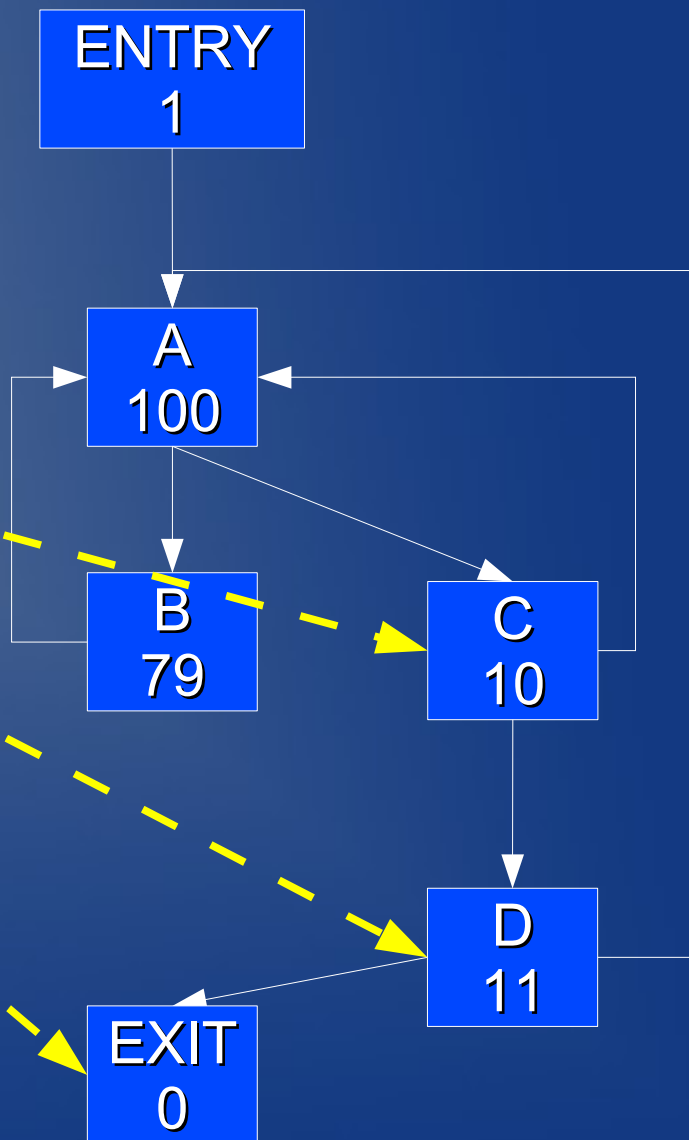
Сбор профиля



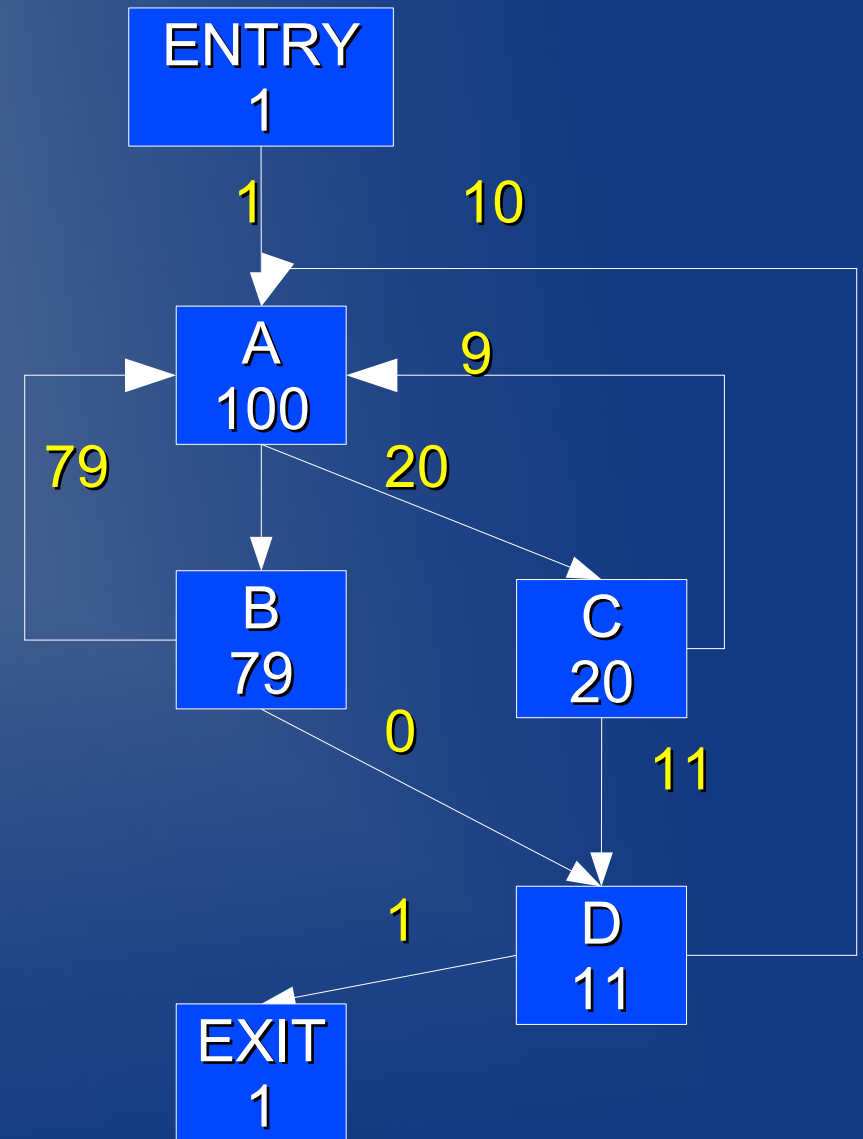
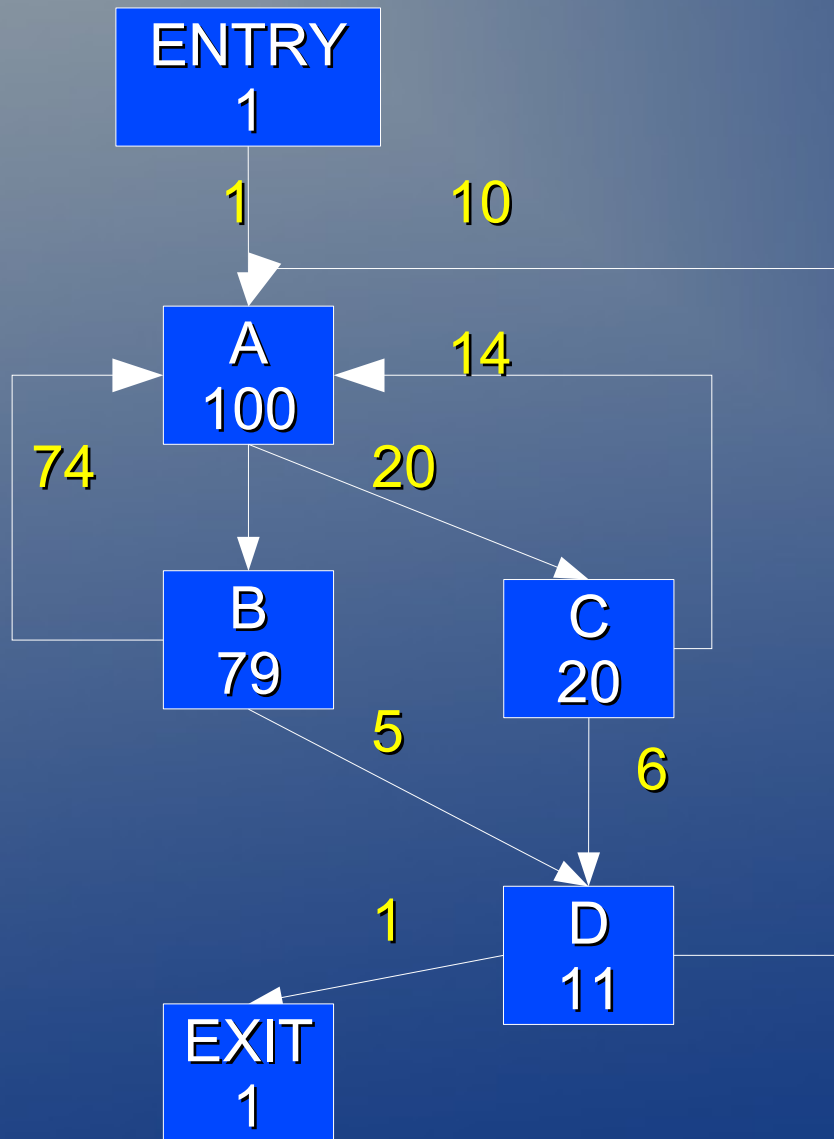
Результаты профилирования

Конфликты!

```
while (1)
{
  A; if (cmp1)
    B;
  else
  {
    C; if (cmp2)
      {
        D; if (cmp3)
          return;
      }
  }
}
```



Восстановление профиля по рёбрам



Интерпретация и JIT

Lib.cpp

```
int min (int x, int y)
{
if (x < y) return x;
else      return x;
}
```

Отрывок байткод тела функции
в Game.bc

43 c5 82 38 35 79 1A B4

Интерпретатор

- 1) Взять очередную команду
- 2) Получить её операнды
- 3) Выполнить
- 4) Перейти к следующей (на шаг 1)

Программу исполняет интерпретатор

JIT

- 1) Скомпилировать байт код в машинный код
- 2) Прыгнуть на этот код

Программу исполняет процессор

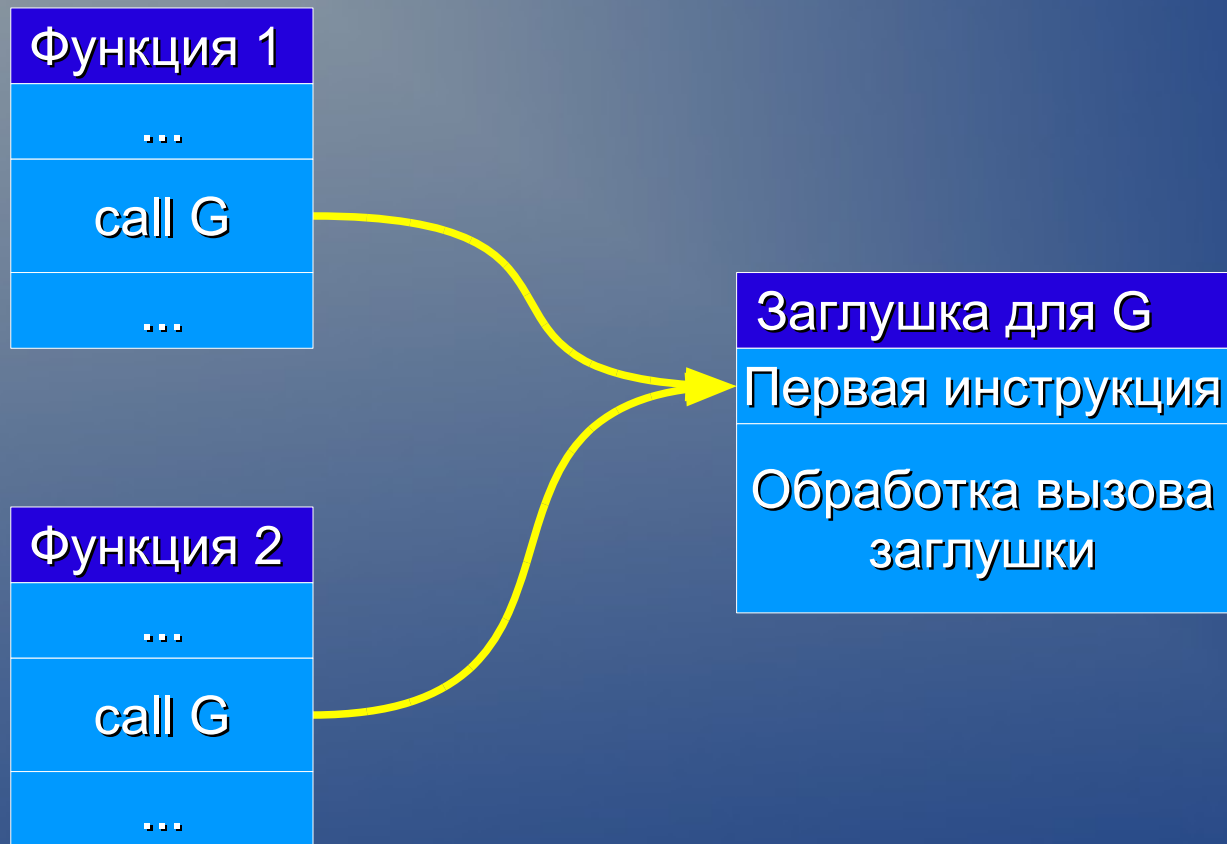
JIT в LLVM

Ленивая кодогенерация (пока не вызовем, не компилируем)

Память под глобальные переменные выделяется только при первом использовании

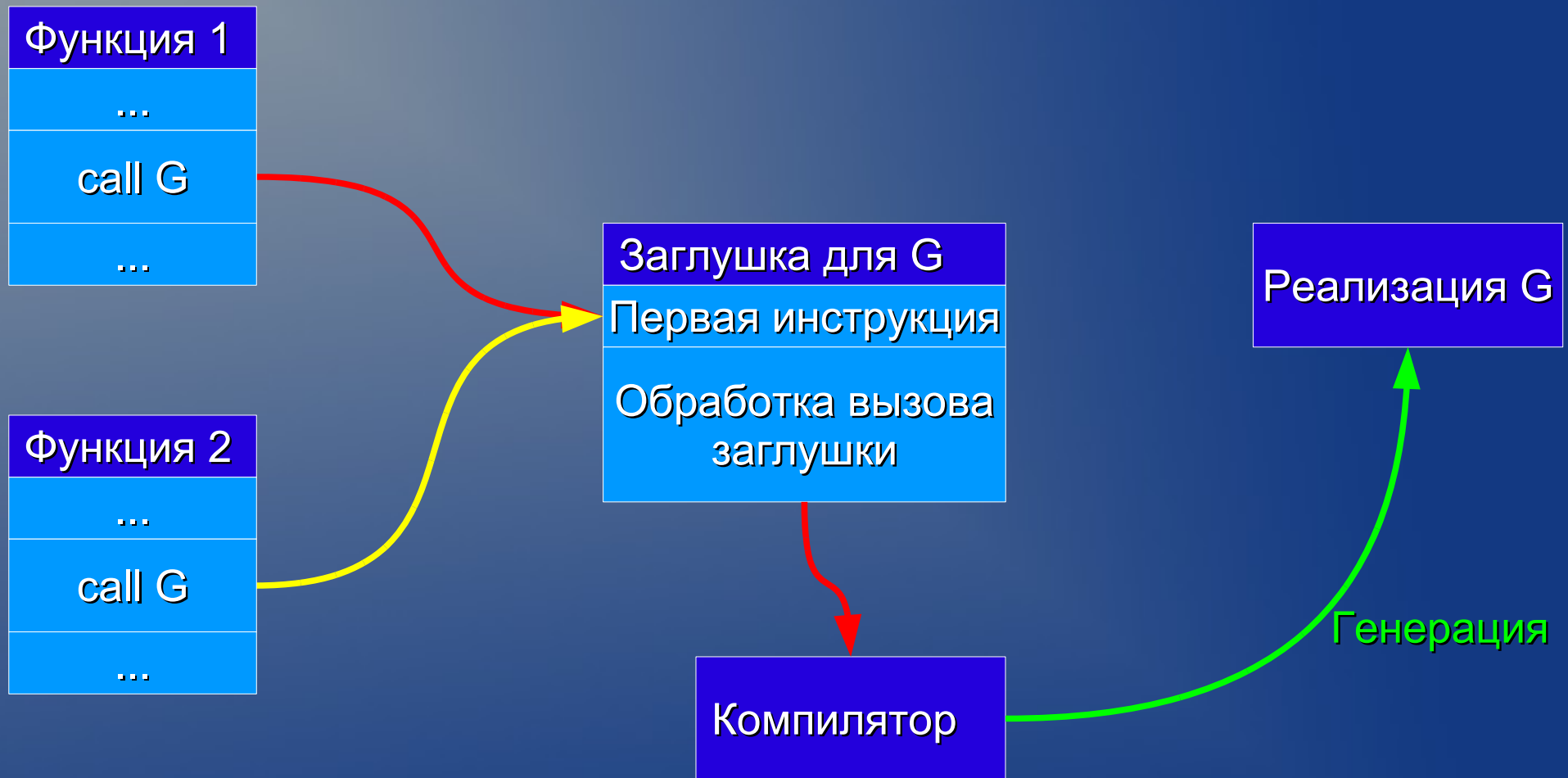
Тело функции может спокойно отсутствовать, если её не вызывать

Реализация заглушек



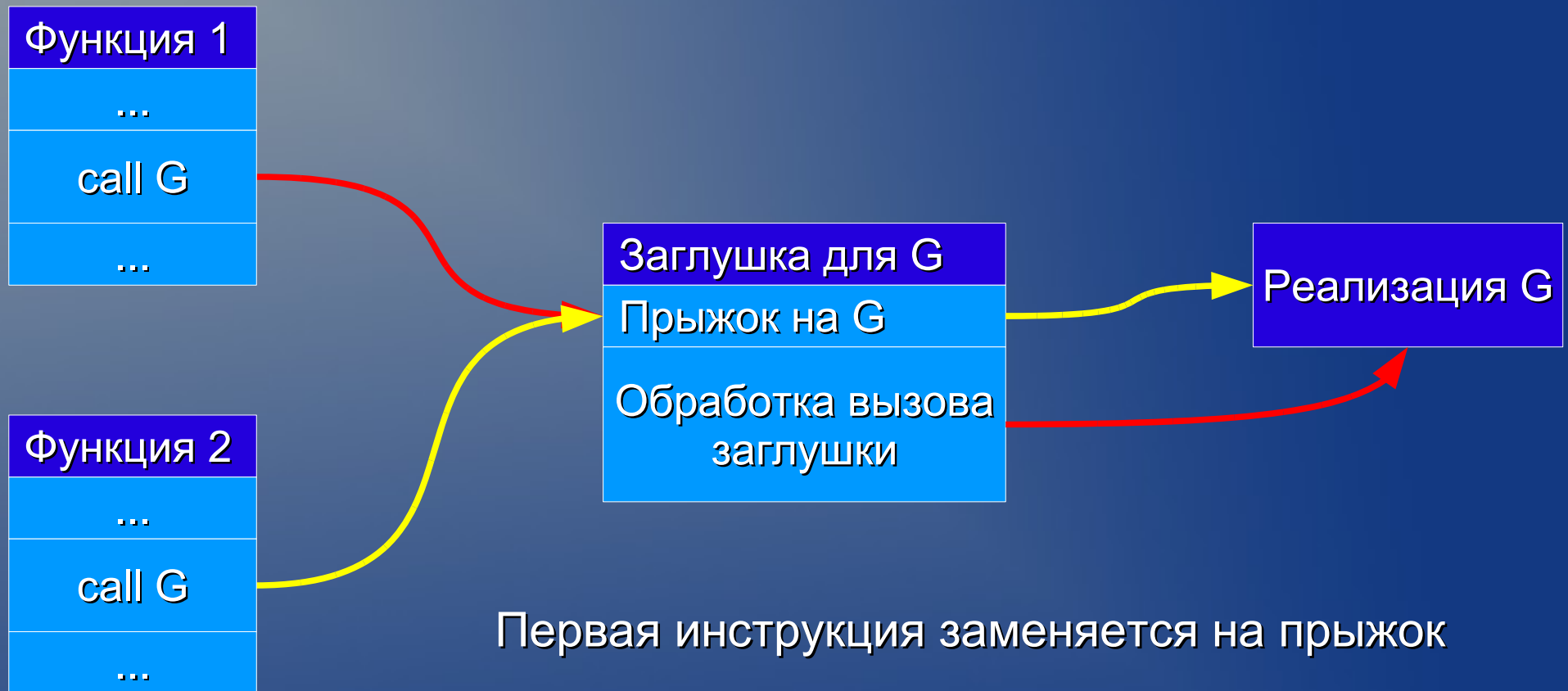
Ни одна из функций пока G не вызывала

Реализация заглушек



Пусть первая функция вызвала G

Реализация заглушек

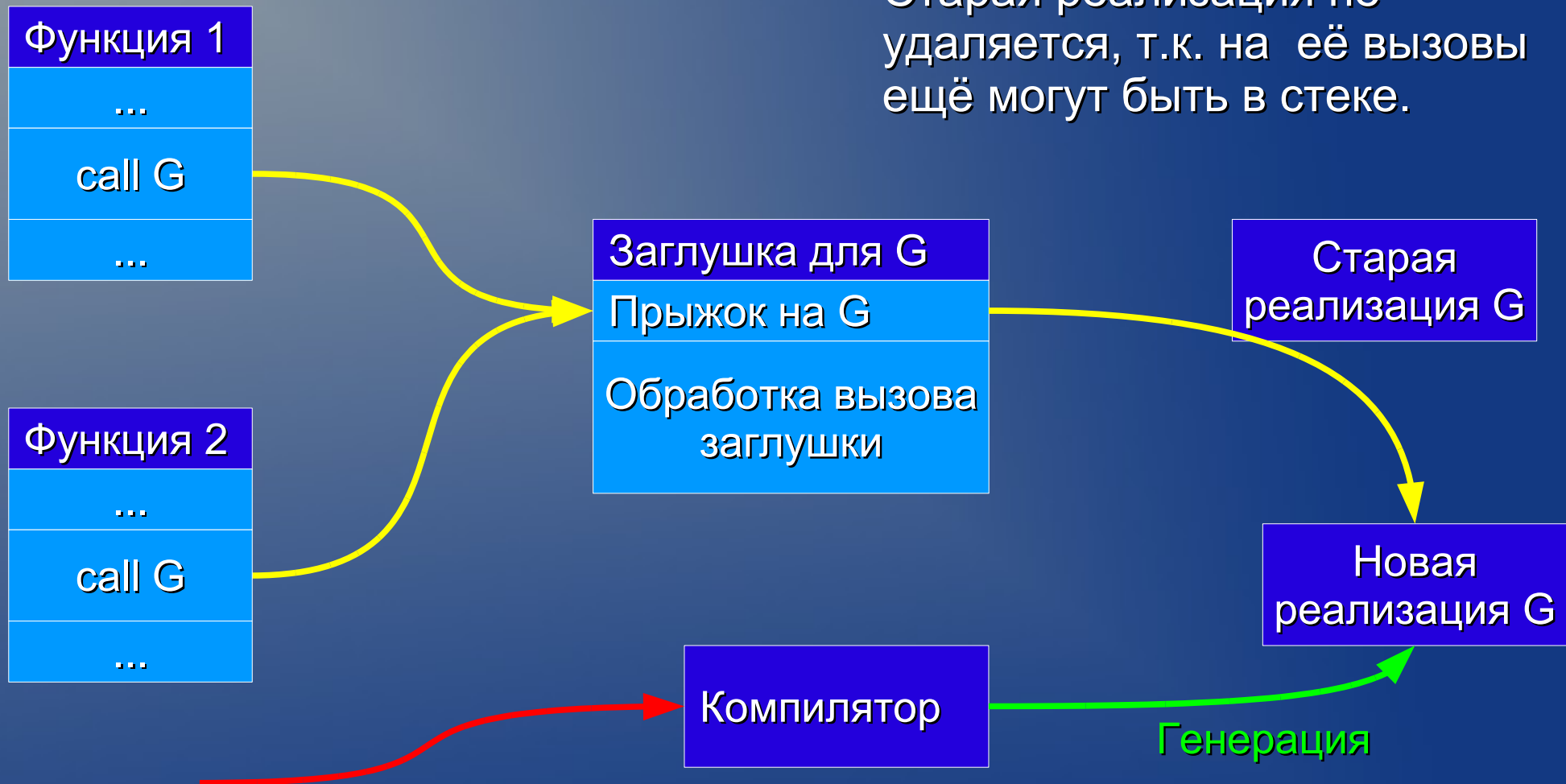


Первая инструкция заменяется на прыжок

Для всего остального кода адрес вызова G не меняется!

Перекомпиляция

Старая реализация не удаляется, т.к. на её вызовы ещё могут быть в стеке.



Создаётся новая функция и переставляется прыжок

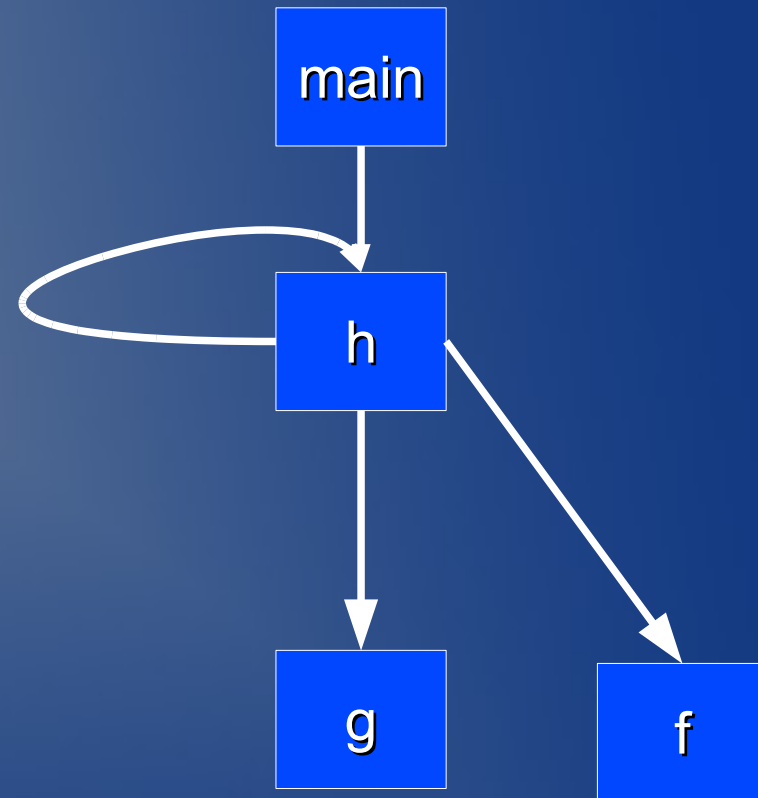
Текущее состояние проекта

Ведутся работы по взаимодействию OProfile и LLVM-драйвера. Они осложнены тем, что последние версии OProfile должны быть частично вкомпилированы в ядро Линукса.

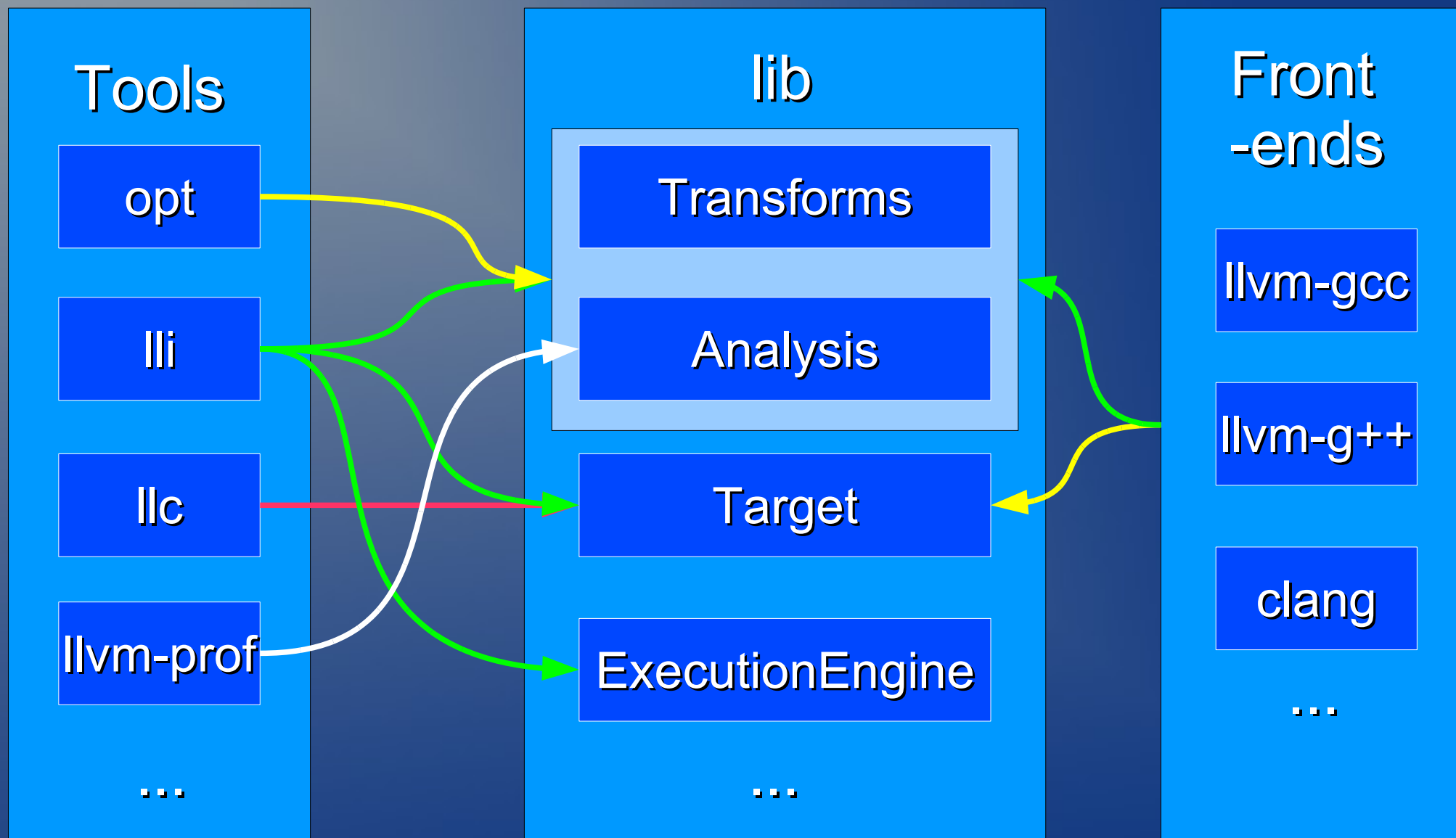
Ведётся поиск и реализация профилезависимых оптимизаций, а так же параметризация существующих. Работа осложняется отсутствием хорошей поддержки профиля в LLVM и автоматического обновления некоторых структур данных LLVM.

Граф ВЫЗОВОВ Call graph

```
void f() {}  
void g() {}  
  
void h(int x) {  
    if (!x)  
        f();  
  
    if (x > 10)  
        h(x - 1);  
  
    g();  
}  
  
int main() {  
    h(100);  
    return 0;  
}
```



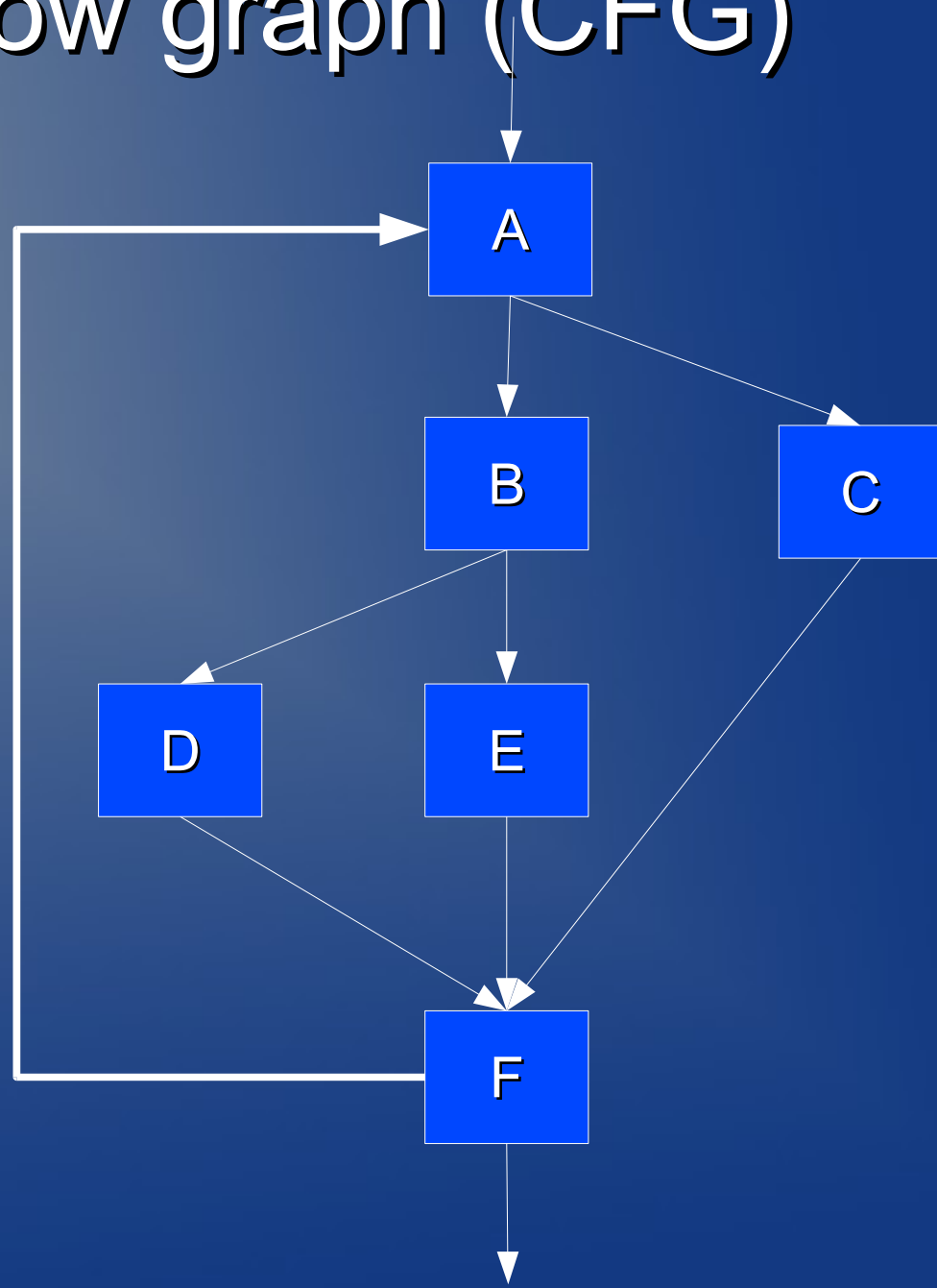
Архитектура LLVM



Граф потока управления

Control flow graph (CFG)

```
void f() {  
  do {  
    A;  
    if (pred1) {  
      B;  
      if (pred2)  
        D;  
      else E;  
    }  
    else C;  
    F;  
  } while (somth);  
}
```



Промежуточное представление

Функция представлена в виде CFG, каждый базовый блок состоит из простых команд

Операции понижаются до трёхадресных команд (результат и 2 операнда)

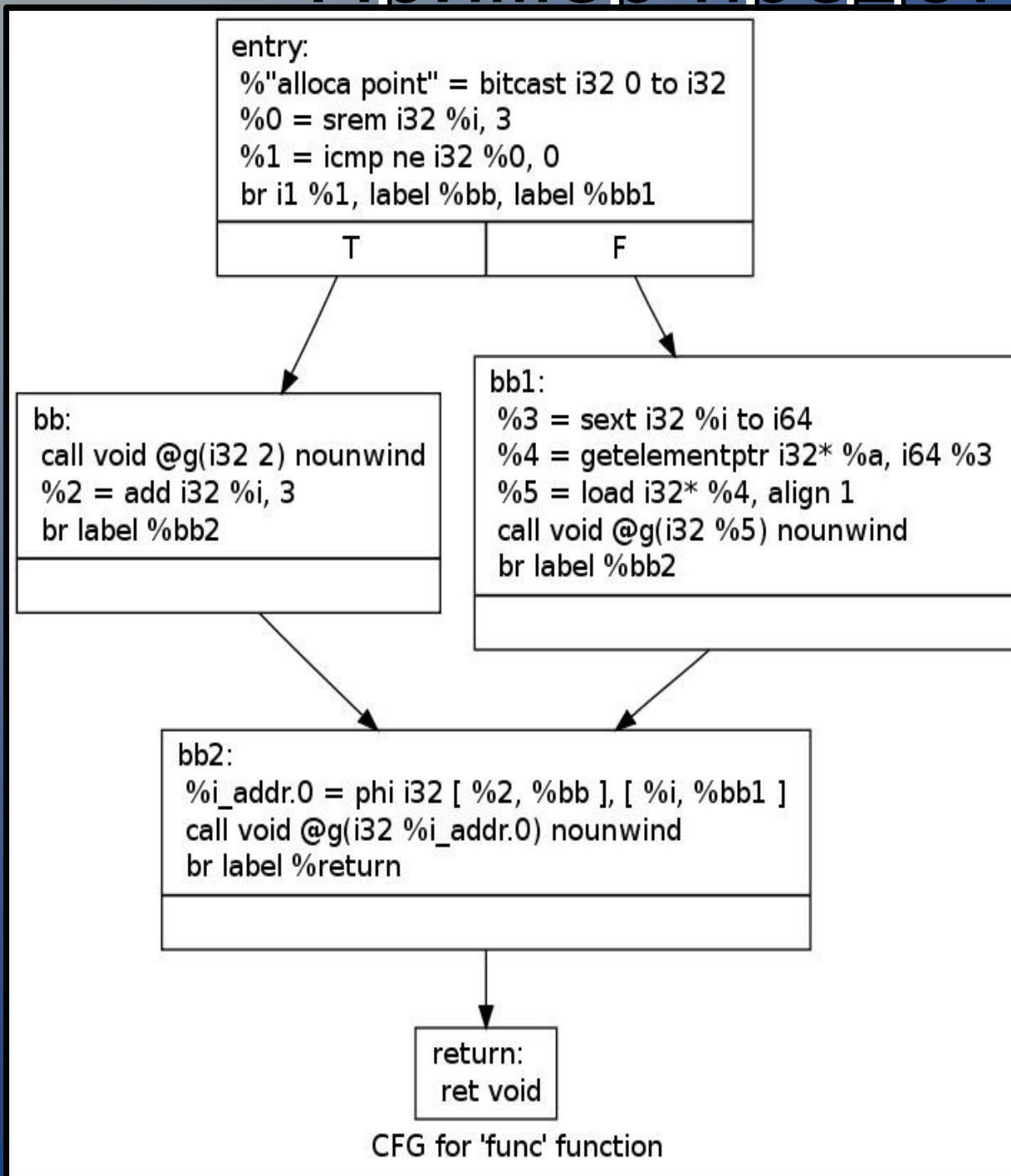
```
%indvar.next17 = add i64 %ix.014, 1  
%8 = load i32* %a_addr, align 4
```

Строгая типизация (если тип не соответствует, производится приведение типа)

Есть два типа вызовов (обычный и с обработчиком исключений)

SSA-представление

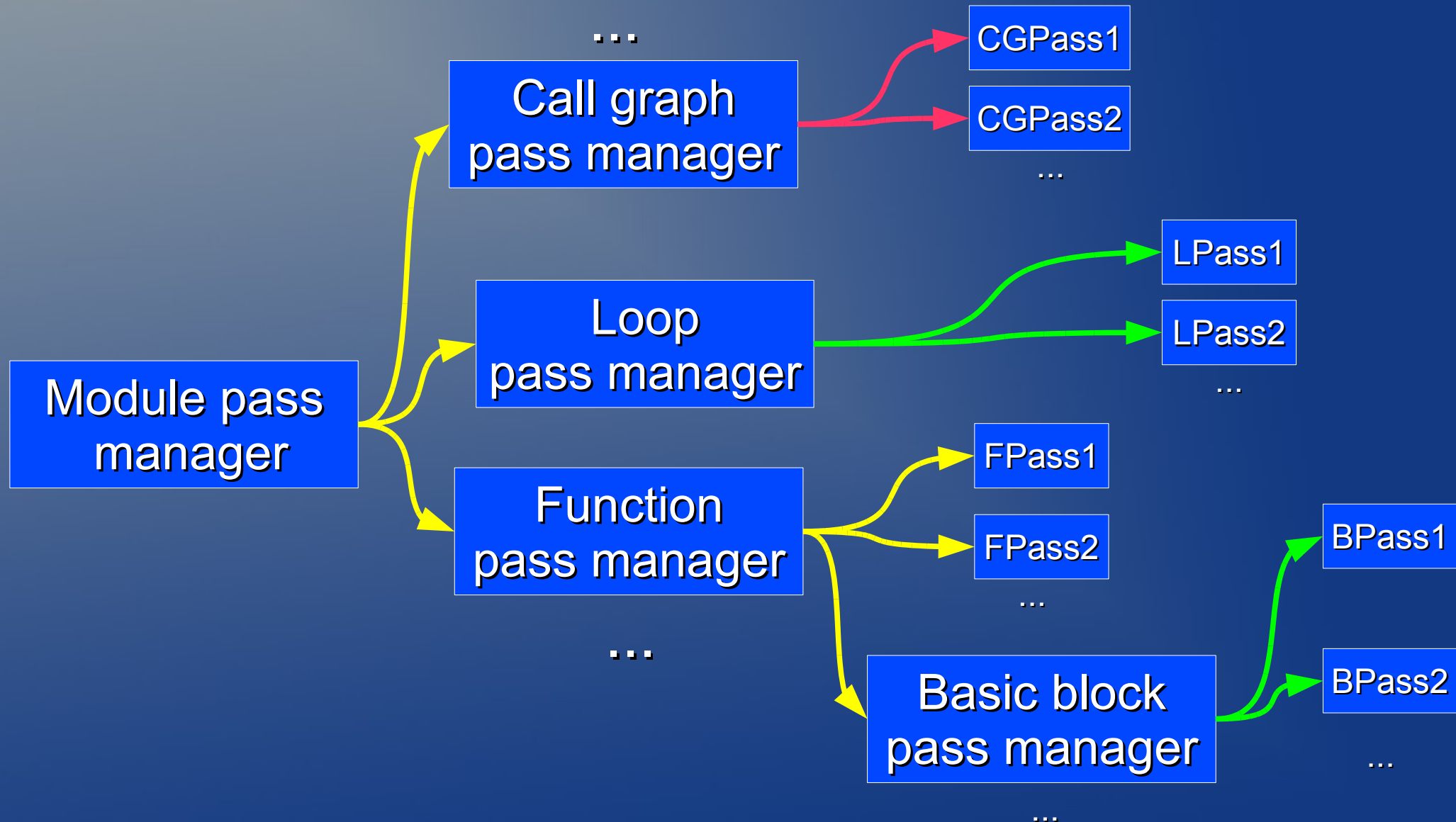
Пример представления



```
void
func (int i, int* a)
{
    if (i % 3)
    {
        g (2);
        i = i + 3;
    }
    else
    {
        g (a[i]);
    }

    g (i);
}
```

Архитектура пассов



Структура пасса

Методы пасса

getAnalysisUsage

doInitialization

run

doFinalization

Типы пассов

Module Pass

Call Graph Pass

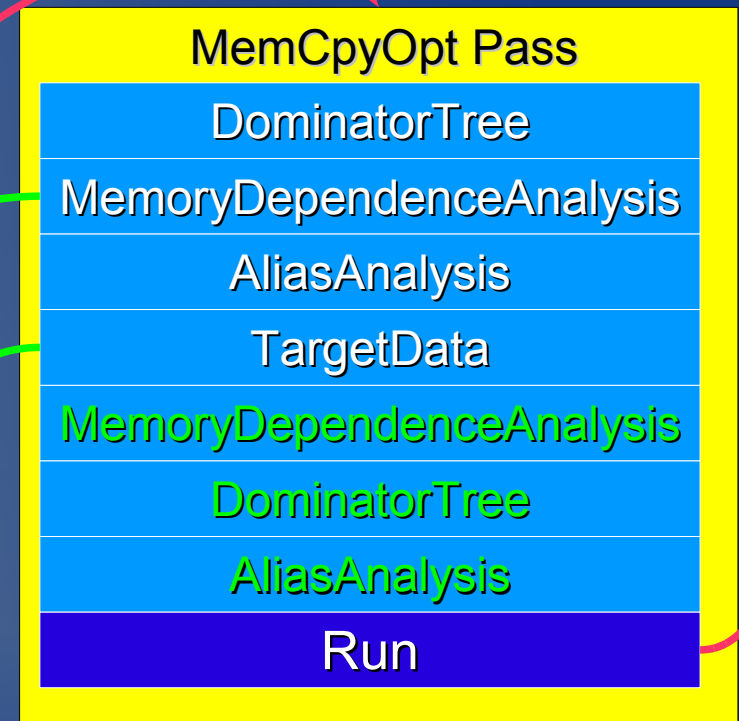
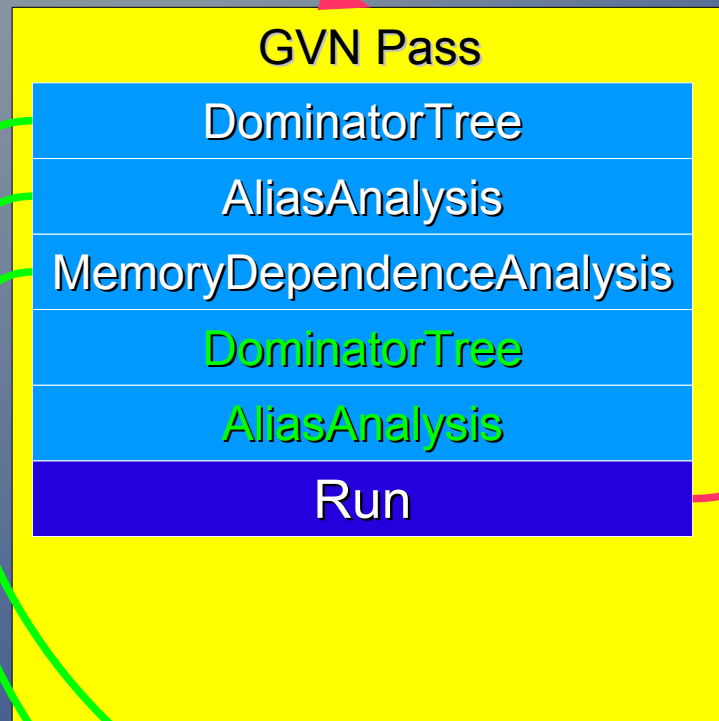
Function Pass

Loop Pass

Basic Block Pass

Machine Function Pass

Проведение оптимизаций



DominatorTree

AliasAnalysis

MemoryDependenceAnalysis

TargetData

Возможности JIT инфраструктуры

Компиляция функций по требованию.

Ленивая компиляция.

Перекомпиляция.

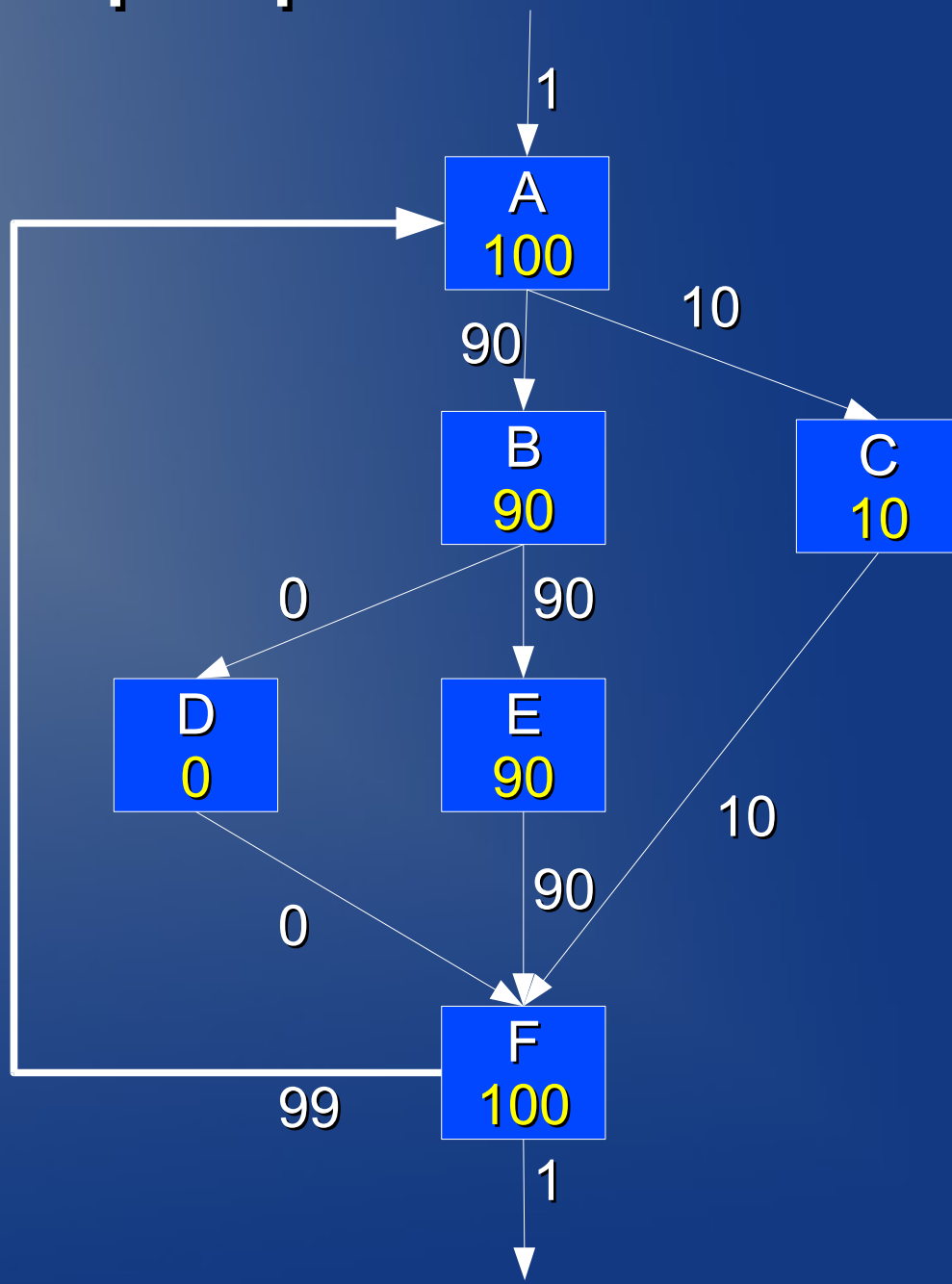
Удаление функции.

Создание или удаление глобальной переменной.

Получение адреса функции или глобальной переменной.

Профиль программы

```
void f() {  
  do {  
    A;  
    if (pred1) {  
      B;  
      if (pred2)  
        D;  
      else E;  
    }  
    else C;  
    F;  
  } while (somth);  
}
```



Что мы хотим от LLVM

Нить программы

Исполнение кода

Первая компиляция
функции

Нить монитора

Профилирование

Перекомпиляция
с учётом
собранных данных

Оптимизации в LLVM, использующие профиль

Basic block reordering

А вот и всё!

Оптимизации в GCC, использующие профиль

Basic block reordering

Tail duplication (формирование суперблоков)

Inlining

Loop unrolling and peeling

Loop unswitching

Instruction scheduling

Branch target register load optimizations

Interprocedural constant propagation

И несколько других

Что хотим реализовать

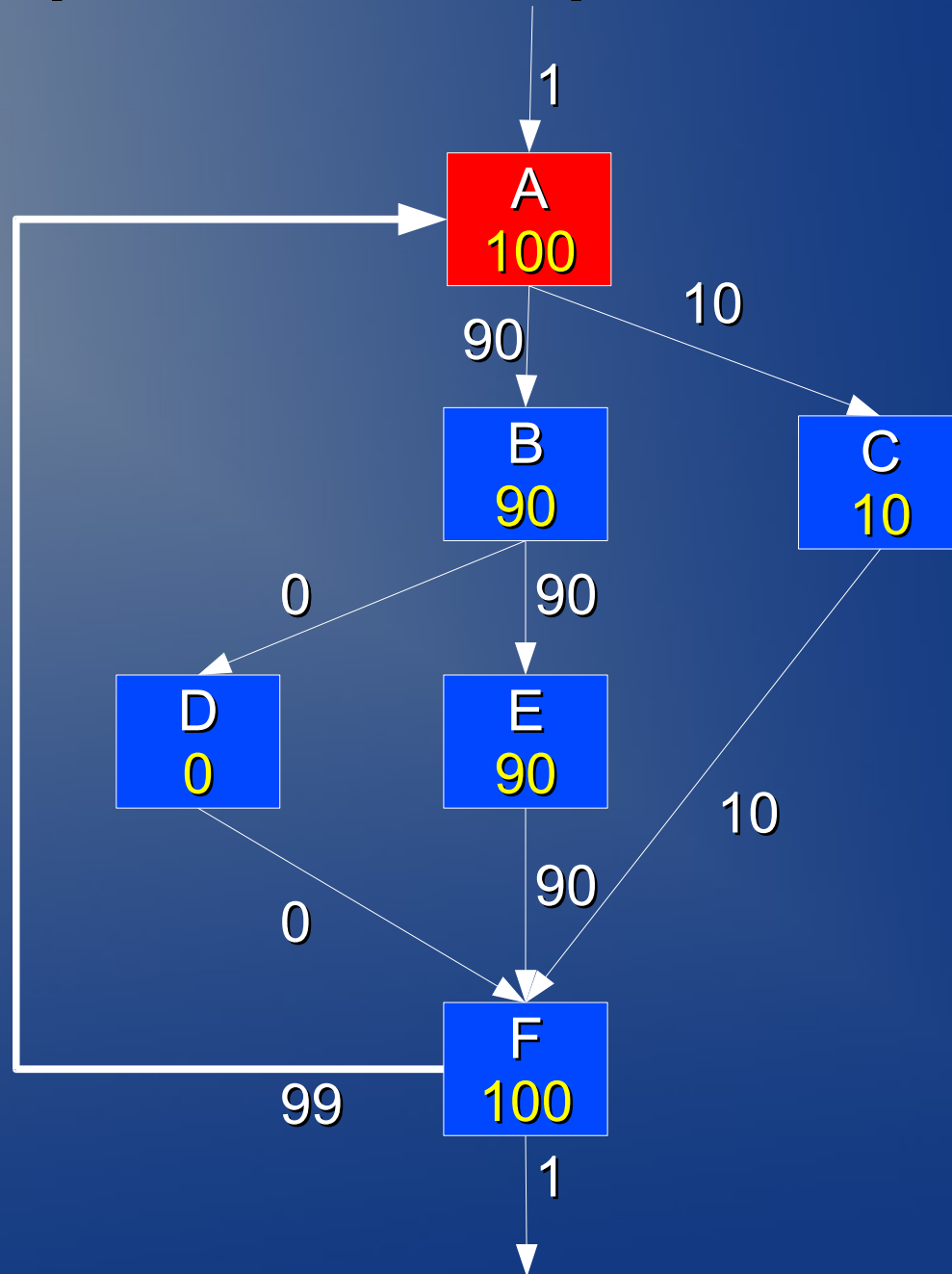
Inlining

Loop unrolling and peeling

The formation of super-blocks

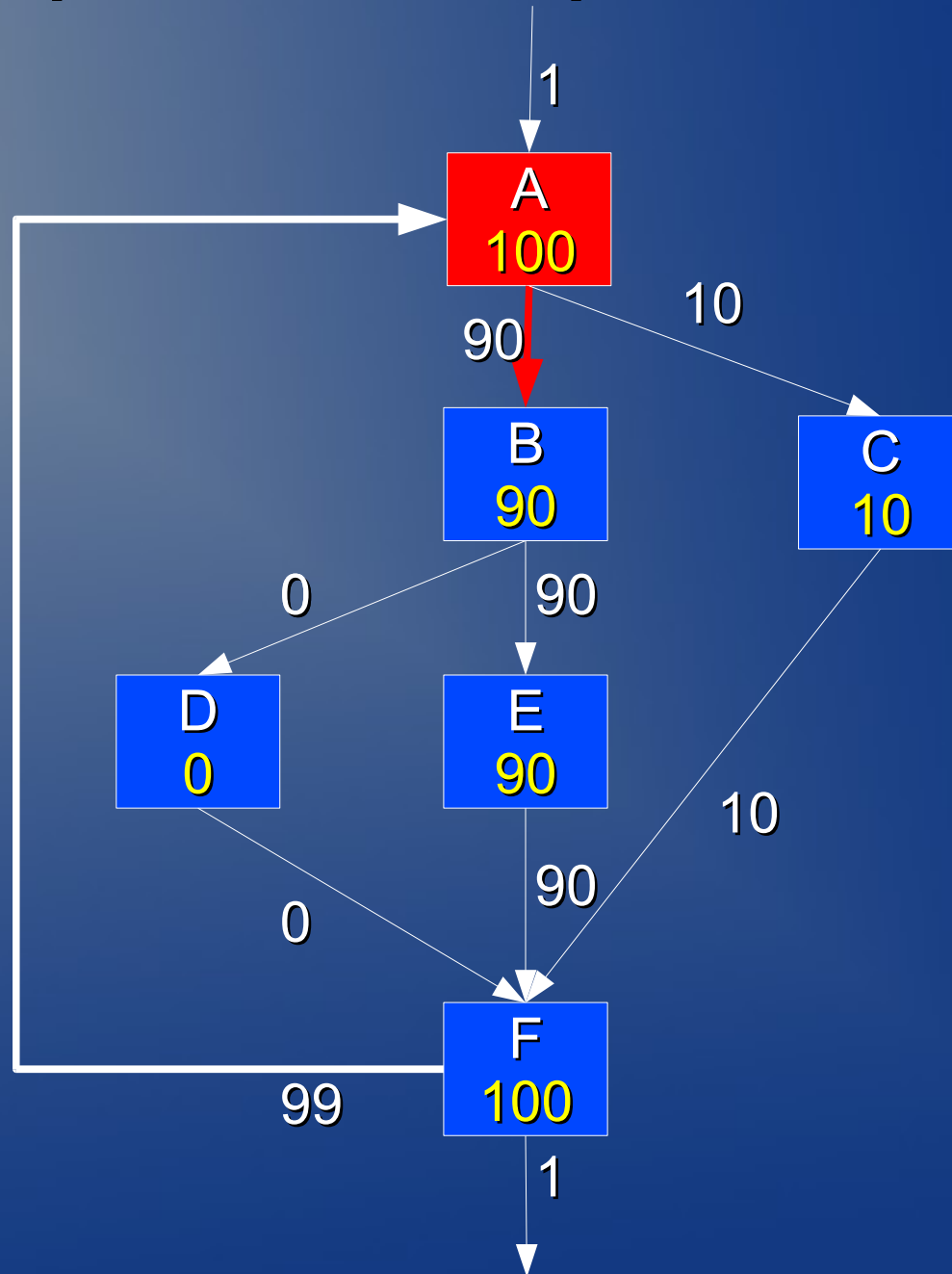
Basic block reordering (уже есть)

Формирование трассы



Находим
самый часто
исполняемый
базовый блок

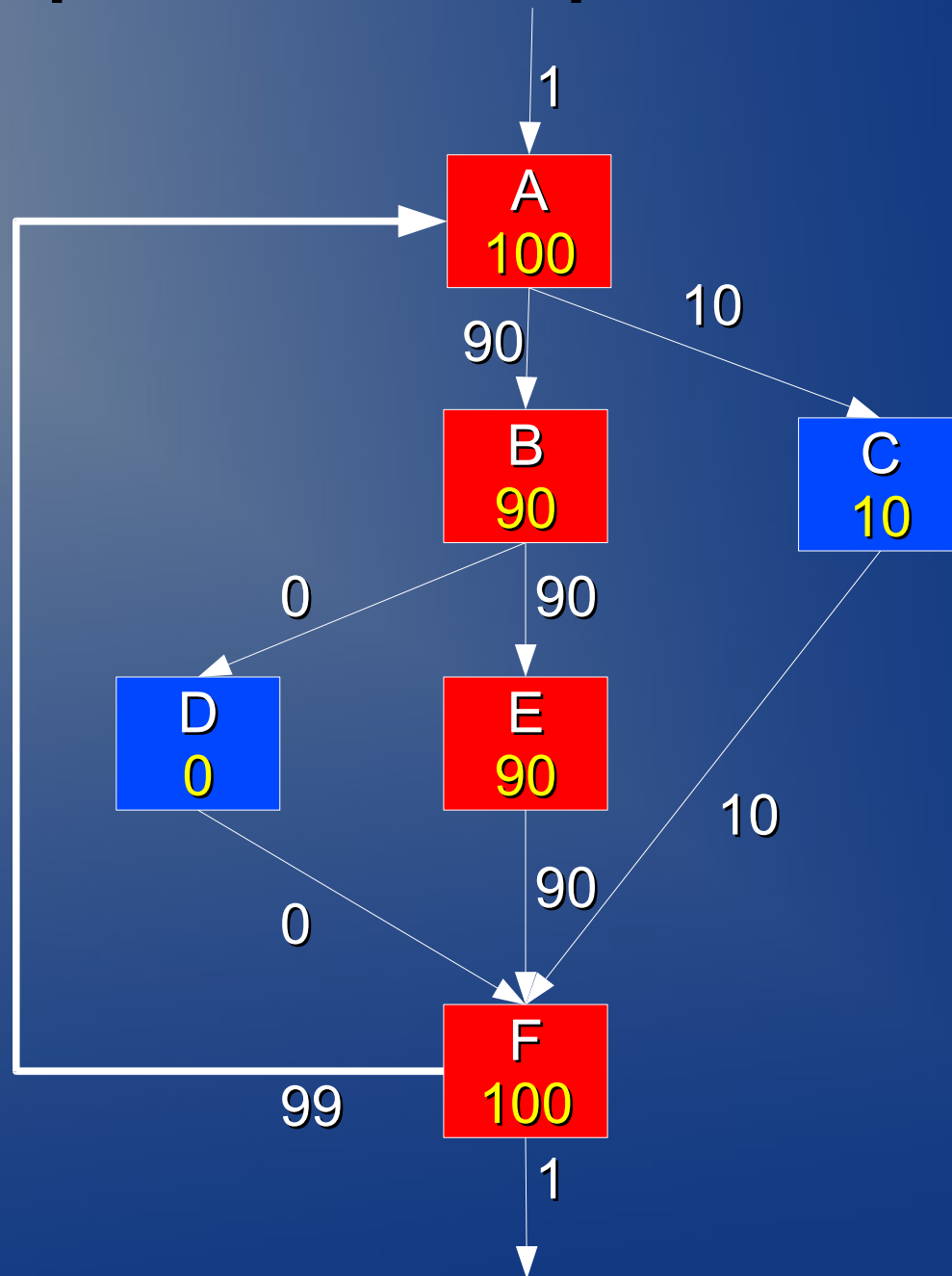
Формирование трассы



Находим
самое часто
исполняемое
ребро

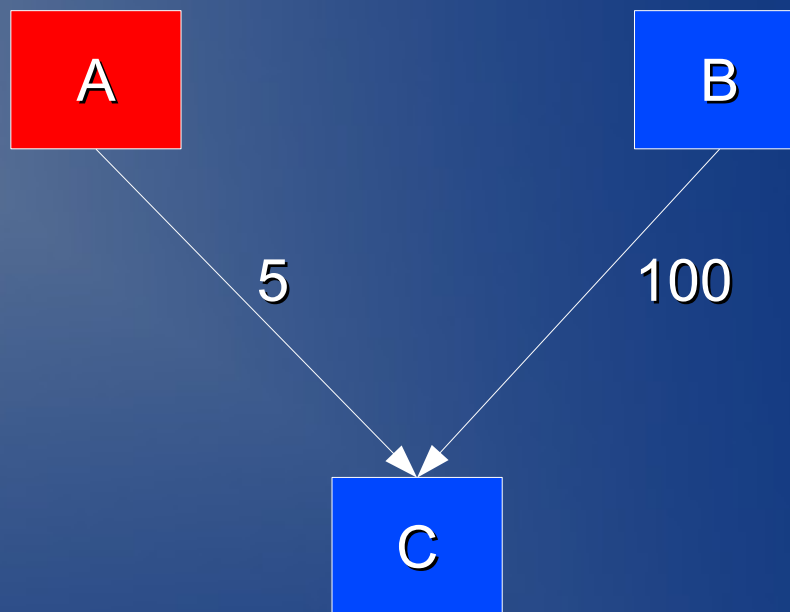
Формирование трассы

Продолжаем процесс до тех пор, пока либо не упрёмся в уже имеющийся в трассе базовый блок, либо базовый блок не будет «подходящим» для этой трассы

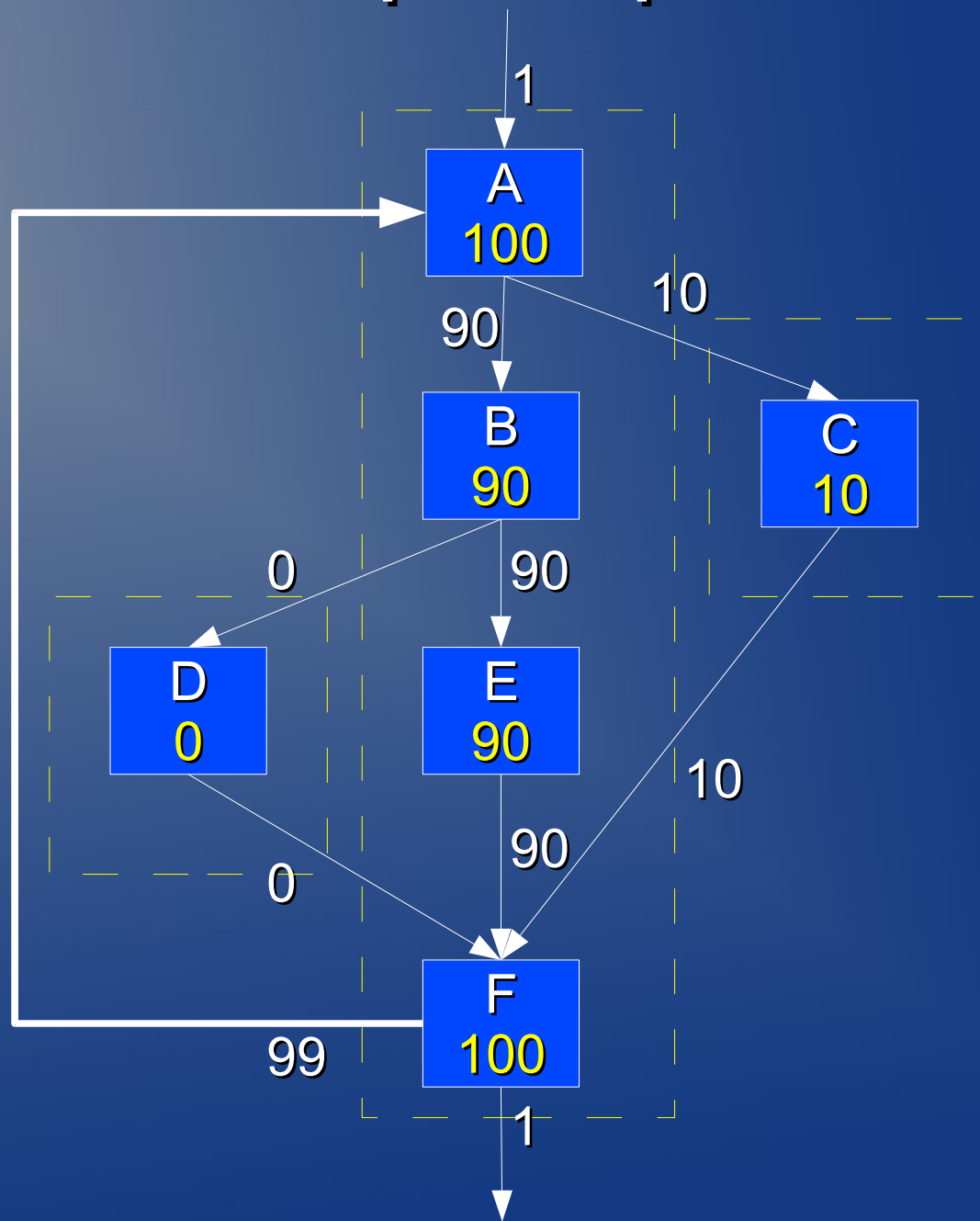


«Неподходящий» базовый блок

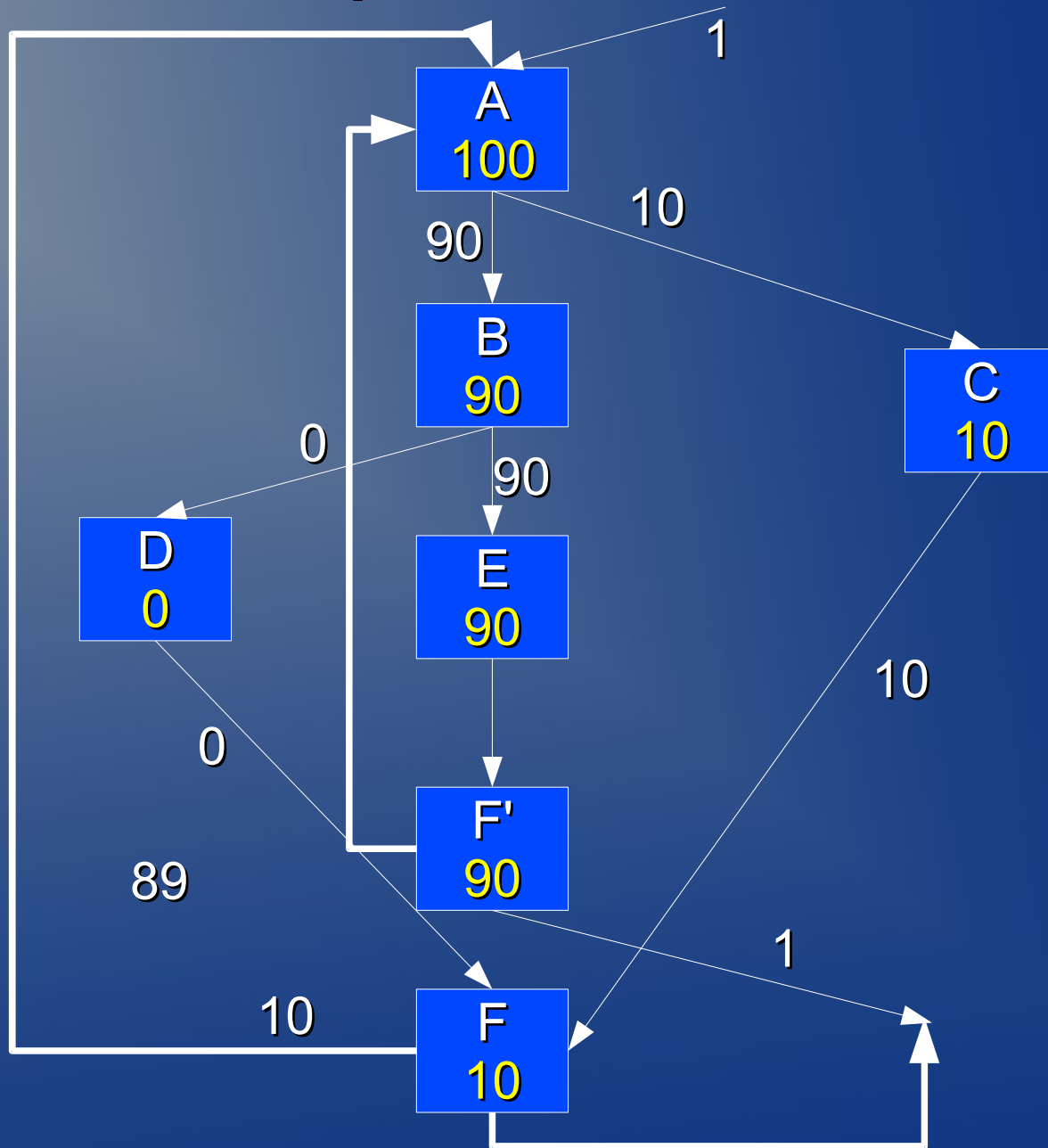
Если для кандидата С
на добавление блок А
будет не лучшим
предшественником, то
не следует включать С
в трассу



Трассы в примере



Tail duplication



Оптимизации, извлекающие пользу от суперблоков

constant propagation (В LLVM есть аж 3 разновидности)

copy propagation

constant combining

common subexpression elimination

redundant store elimination (уже есть в LLVM)

redundant load elimination (уже есть в LLVM)

dead code removal (уже есть в LLVM)

loop invariant code removal (уже есть в LLVM)

loop induction variable elimination

Что сделано

Разделение на 2 потока (в одном программа, в другом — монитор и перекомпиляция).

Исправлен баг с большими глобальными переменными.

При наличие в программе больших глобальных массивов происходило переполнение буфера во время JIT'a и LLVM падал. Проблема решилась перевыделением большего куска памяти.

Реализована попытка перекомпиляции самых горячих функций.

Данная стратегия позволила в большинстве случаев усреднить время между JIT'овскими O0 и O2

Что сделано (продолжение)

Реализован маппинг данных по инструкциям семплинга в счётчики базовых блоков.

Во время генерации инструкций очередного MachineBasicBlock добавляется пара (адрес первой инструкции MachineBasicBlock, соответствующий данному MachineBasicBlock'у BasicBlock).

Переделана инфраструктура работы с профилем

Раньше профиль хранился в виде отдельной структуры данных, которую никто не обновлял и мог использовать только один пасс. Теперь счётчики интегрированы в соответствующие структуры и они могут обновляться вспомогательными процедурами.

Идёт разработка пасса по формированию суперблоков.

Основная проблема в том, что SSA представление приходится обновлять вручную, либо повышать представление, а потом заного его понижать

Оптимизации в Java JIT, использующие профиль

Feedback-directed method inlining

Feedback-directed code positioning

Feedback-directed loop unrolling

Instruction scheduling

Region-based inlining