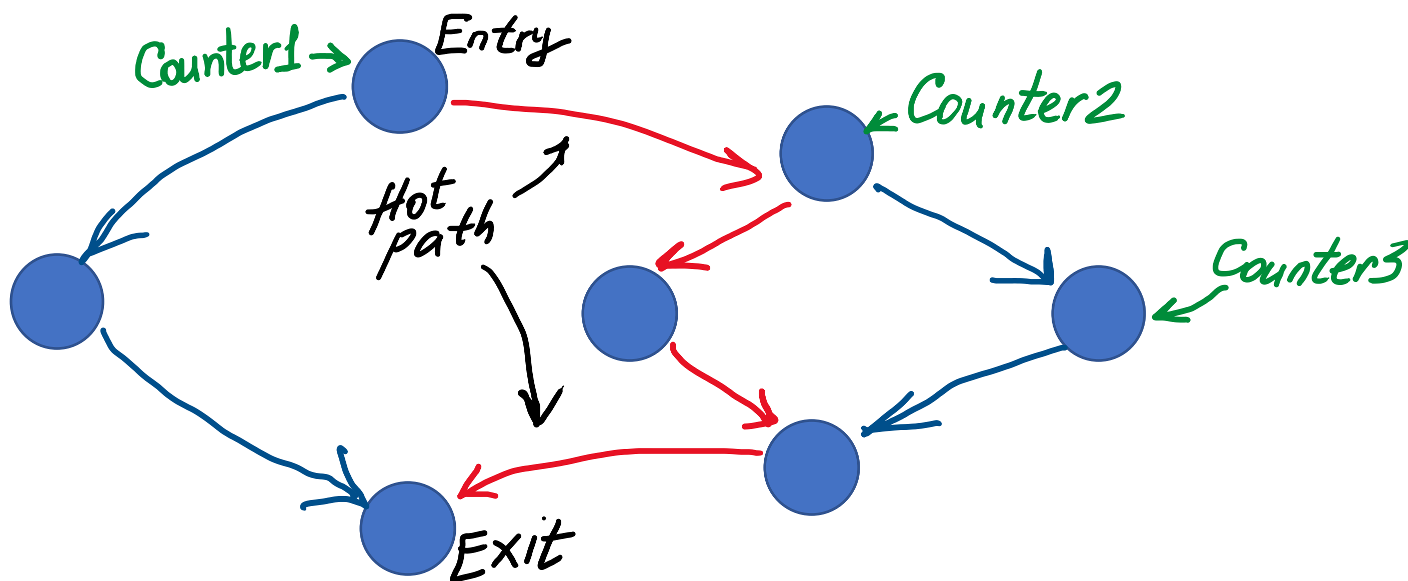
PGO:
Как устроено
и как использовать

Оптимизация с использованием профиля выполнения программы

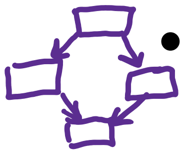


Коротко об авторе

- Разработка тулчейнов (компиляторы/отладчики/загрузчики)
  - https://github.com/openharmony

Коротко об авторе

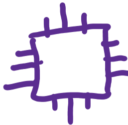
- Разработка тулчейнов (компиляторы/отладчики/загрузчики)
  - https://github.com/openharmony



- Улучшение Clang/LLVM
  - https://reviews.llvm.org/people/revisions/18946/

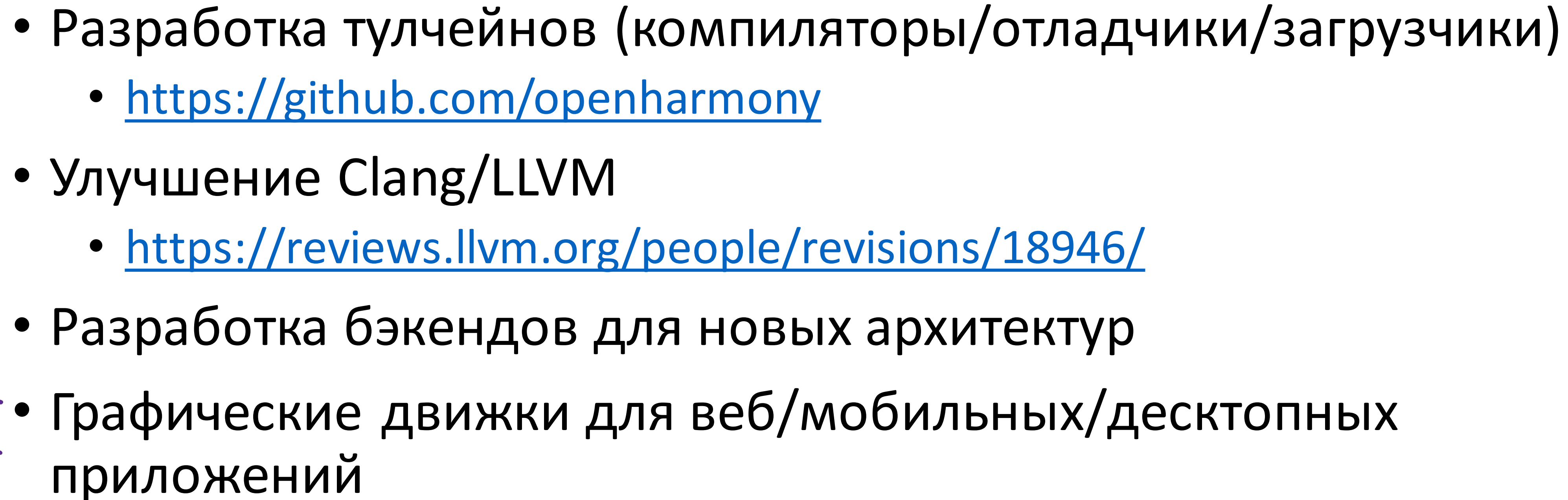
Коротко об авторе

- Разработка тулчейнов (компиляторы/отладчики/загрузчики)
  - https://github.com/openharmony
- Улучшение Clang/LLVM
  - https://reviews.llvm.org/people/revisions/18946/



- Разработка бэкендов для новых архитектур

Коротко об авторе



- Разработка тулчейнов (компиляторы/отладчики/загрузчики)
  - https://github.com/openharmony
- Улучшение Clang/LLVM
  - https://reviews.llvm.org/people/revisions/18946/
- Разработка бэкендов для новых архитектур
- Графические движки для веб/мобильных/десктопных приложений

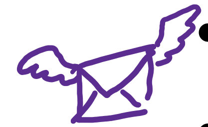
Коротко об авторе

- Разработка тулчейнов (компиляторы/отладчики/загрузчики)
  - https://github.com/openharmony
- Улучшение Clang/LLVM
  - https://reviews.llvm.org/people/revisions/18946/
- Разработка бэкендов для новых архитектур
- Графические движки для веб/мобильных/десктопных приложений
- Функциональное программирование / Rust

$fx = x$

Коротко об авторе

- Разработка тулчейнов (компиляторы/отладчики/загрузчики)
  - https://github.com/openharmony
- Улучшение Clang/LLVM
  - https://reviews.llvm.org/people/revisions/18946/
- Разработка бэкендов для новых архитектур
- Графические движки для веб/мобильных/десктопных приложений
- Функциональное программирование / Rust



- Подробнее: Linkedin
- Связь kpdev42@gmail.com / tg:@kpdev42

PGO. Знакомство

- **PGO (aka FDO, aka PDF, aka POGO) - Profile-Guided Optimization:** Оптимизация с использованием профиля выполнения программы. Это не оптимизация. Это подход к оптимизации. Поддерживается всеми распространенными компиляторами (Clang, GCC, Intel, MSVC, ...)

PGO. Знакомство

- **PGO (aka FDO, aka PDF, aka POGO) - Profile-Guided Optimization:** Оптимизация с использованием профиля выполнения программы. Это не оптимизация. Это подход к оптимизации. Поддерживается всеми распространенными компиляторами (Clang, GCC, Intel, MSVC, ...)
- **Профиль программы:** данные о выполнении программы. Какие функции использовались, по каким путям проходило их выполнение

PGO. Знакомство

- **PGO (aka FDO, aka PDF, aka POGO) - Profile-Guided Optimization:** Оптимизация с использованием профиля выполнения программы. Это не оптимизация. Это подход к оптимизации. Поддерживается всеми распространенными компиляторами (Clang, GCC, Intel, MSVC, ...)
- **Профиль программы:** данные о выполнении программы. Какие функции использовались, по каким путям проходило их выполнение
- **Сценарий использования программы:** набор(ы) входных данных (в т.ч. пользовательского ввода), обработку которого мы хотим оптимизировать

PGO. Применение

- Google Chrome (детали оптимизации), Firefox (детали оптимизации). Бенчмарк Speedometer (Link) показывает ускорение до …%

PGO. Применение

- Google Chrome (детали оптимизации), Firefox (детали оптимизации). Бенчмарк Speedometer (описание) показывает ускорение до 12%

PGO. Применение

- Google Chrome (детали оптимизации), Firefox (детали оптимизации). Бенчмарк Speedometer (описание) показывает ускорение до 12%

- Clang. Уменьшение времени компиляции до ...%. Детали

PGO. Применение

- Google Chrome (детали оптимизации), Firefox (детали оптимизации). Бенчмарк Speedometer (описание) показывает ускорение до 12%

- Clang. Уменьшение времени компиляции до 20%. Детали

PGO. Применение

- Google Chrome (детали оптимизации), Firefox (детали оптимизации). Бенчмарк Speedometer (описание) показывает ускорение до 12%
- Clang. Уменьшение времени компиляции до 20%. Детали
- Linux Kernel. Оптимизация ядра приводит к ускорению приложений (до 10% в исследовании), которые интенсивно используют системные вызовы (базы данных, вебсервера и т.д.).

Профиль программы

- Описание выполнения программы. Есть много форматов этого описания, например:
  - Perfdata / Simpleperf (extended perfdata)
  - LLVM-profdata
  - GCC .gcda files
  - ...
- Способы получения профиля:
  - Инструментация
  - Семплирование

Инструментация (на примере LLVM)

- Вставка дополнительных инструкций в код программы, обеспечивающих запись следующих параметров:
    - Поток выполнения внутри функции
    - Адреса косвенных вызов
    - Аргументы функций работы с памятью

Инструментация (на примере LLVM)

- Вставка дополнительных инструкций в код программы, обеспечивающих запись следующих параметров:
  - Поток выполнения внутри функции
  - Адреса косвенных вызов
  - Аргументы функций работы с памятью
- Виды инструментации:
  - На уровне фронтенда (FE-level)
  - На уровне промежуточного представления LLVM (IR-level)

Инструментация (на примере LLVM)

- Вставка дополнительных инструкций в код программы, обеспечивающих запись следующих параметров:
  - Поток выполнения внутри функции
  - Адреса косвенных вызов
  - Аргументы функций работы с памятью
- Виды инструментации:
  - На уровне фронтенда (FE-level)
  - На уровне промежуточного представления LLVM (IR-level)
- Накладные расходы на инструментацию - зависит от структуры программы: чем больше ветвлений, а особенно операций с памятью и виртуальных вызовов, тем больше просадка по производительности (в среднем от x2 до x10)

LLVM-IR инструментация

1. Сборка инструментированной версии

```
> clang++ -O2 -fprofile-generate code.cc -o code
```

2. Запуск инструментированной версии

```
> LLVM_PROFILE_FILE="code-%p.profraw" ./code
```

3. Объединение профилей и конвертирование их в формат, ожидаемый компилятором

```
> llvm-profdata merge -output=code.profdata code-*.profraw
```

4. Сборка версии, оптимизированной с использованием профиля

```
> clang++ -O2 -fprofile-use=code.profdata code.cc -o code
```

LLVM_PROFILE_FILE модификаторы

- %h - имя хоста
- %p - идентификатор процесса
- %m - уникальный идентификатор бинарного файла
- %t - значение переменной окружения TMPDIR
- %c - обновленные счетчики постоянно записываются в файл (вместо того, чтобы записаться один раз по завершению выполнения программы)

llvm-profdata

- **show:** показывает общую информацию о данных, содержащихся в профиле

```
$ llvm-profdata show ./merged.profdata -all-functions --counts –memop-sizes
Counters:
  main:
    Hash: 0x0000000acc909904
    Counters: 3
    Number of Memory Intrinsics Calls: 1
    Block counts: [50, 11, 1]
    Memory Intrinsic Size Results:
        [  0,  257, 9 ] (100.00%)
Instrumentation level: IR
...
```

- -fprofile-list
- https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization

llvm-profdata

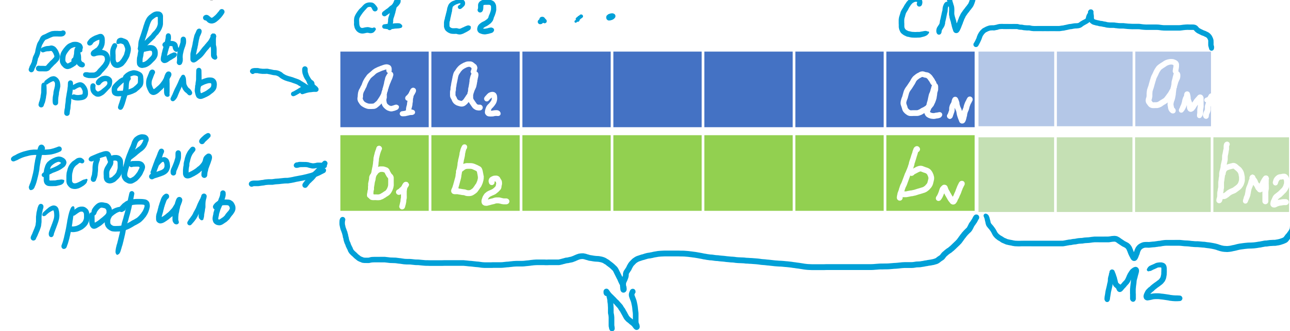
- **overlap:** показывает степень похожести профилей

```
$ llvm-profdata overlap ./first.profdata ./second.profdata
Profile overlap infomation for base_profile: ./first.profdata  and test_profile: ./second.profdata
Program level:
  # of functions overlap: 7
  Edge profile overlap: 84.545%
  Edge profile base count sum: 30
  Edge profile test count sum: 33
  MemOP profile overlap: 83.333%
  MemOP profile base count sum: 9
  MemOP profile test count sum: 6
```

llvm-profdata

- **overlap:** показывает степень похожести профилей

```
$ llvm-profdata overlap ./first.profdata ./second.profdata
Profile overlap infomation for base_profile: ./first.profdata  and test_profile: ./second.profdata
Program level:
  # of functions overlap: 7
  Edge profile overlap: 84.545%
  Edge profile base count sum: 30
  Edge profile test count sum: 33
  MemOP profile overlap: 83.333%
  MemOP profile base count sum: 9
  MemOP profile test count sum: 6
```



llvm-profdata

- **overlap:** показывает степень похожести профилей

```
$ llvm-profdata overlap ./first.profdata ./second.profdata
Profile overlap infomation for base_profile: ./first.profdata  and test_profile: ./second.profdata
Program level:
  # of functions overlap: 7
  Edge profile overlap: 84.545%
  Edge profile base count sum: 30
  Edge profile test count sum: 33
  MemOP profile overlap: 83.333%
  MemOP profile base count sum: 9
  MemOP profile test count sum: 6
```

$$Sum1 = \sum_{i=1}^{N+M1} a_i$$

$$Sum2 = \sum_{i=1}^{N+M2} b_i$$

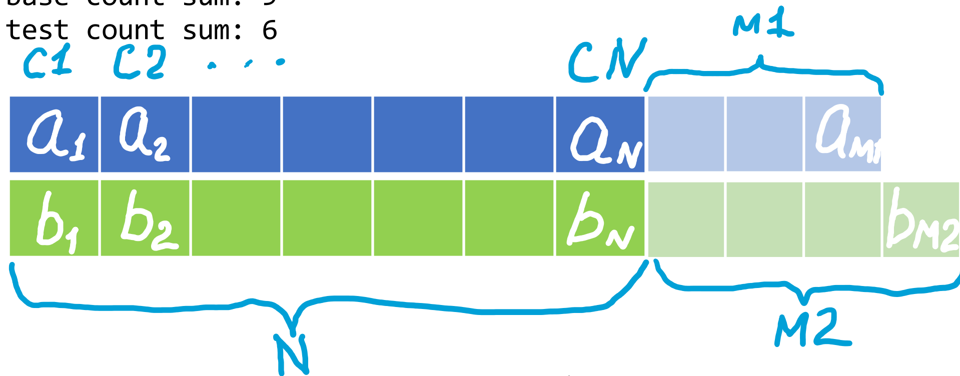
llvm-profdata

- **overlap:** показывает степень похожести профилей

```
$ llvm-profdata overlap ./first.profdata ./second.profdata
Profile overlap infomation for base_profile: ./first.profdata  and test_profile: ./second.profdata
Program level:
  # of functions overlap: 7
  Edge profile overlap: 84.545%
  Edge profile base count sum: 30
  Edge profile test count sum: 33
  MemOP profile overlap: 83.333%
  MemOP profile base count sum: 9
  MemOP profile test count sum: 6
```

Базовый профиль

Тестовый профиль



$$Sum1 = \sum_{i=1}^{N+M1} a_i$$

$$Sum2 = \sum_{i=1}^{N+M2} b_i$$

$$Overlap = \sum_{i=1}^{N} \min\left(\frac{a_i}{Sum1}, \frac{b_i}{Sum2}\right)$$

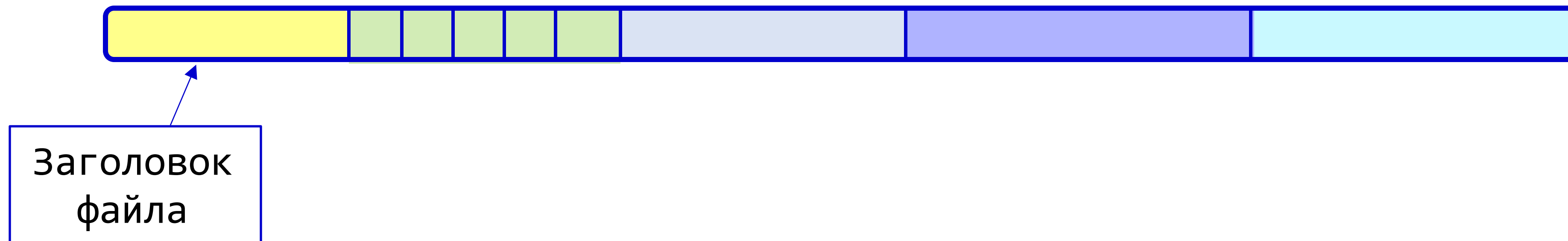
llvm-profdata

- **merge:** объединяет профили и преобразует их в формат, ожидаемый компилятором

```
$ llvm-profdata merge ./*.profraw -o merged.profdata
```

Формат файла с описанием профильной информации

Профиль, полученный после выполнения программы (profraw)



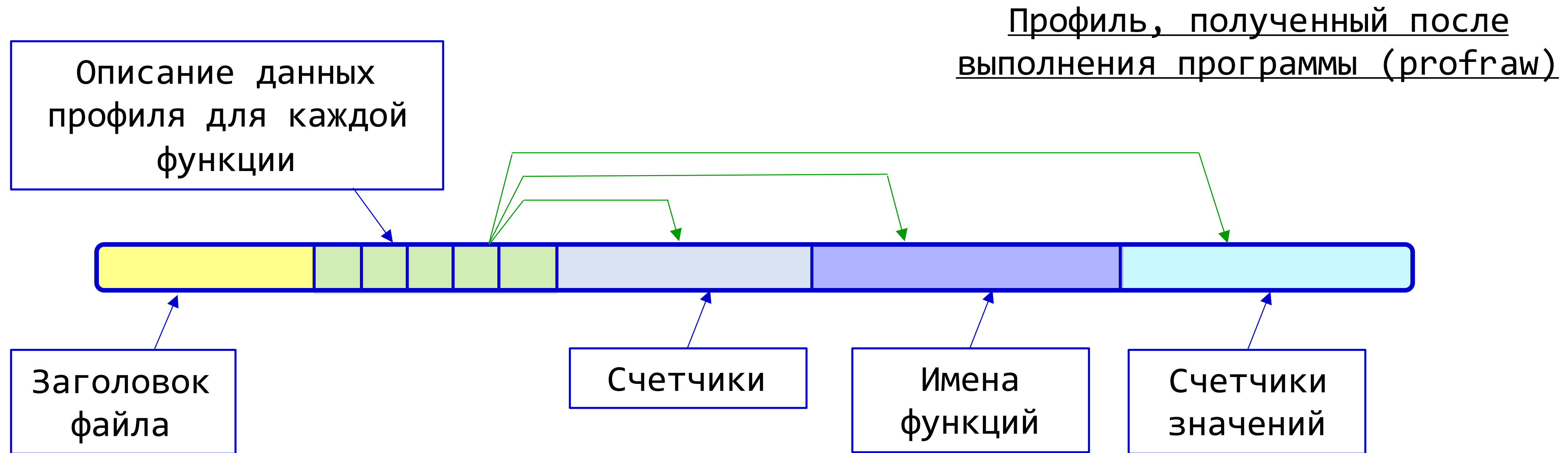
Формат файла с описанием профильной информации

Профиль, полученный после выполнения программы (profraw)

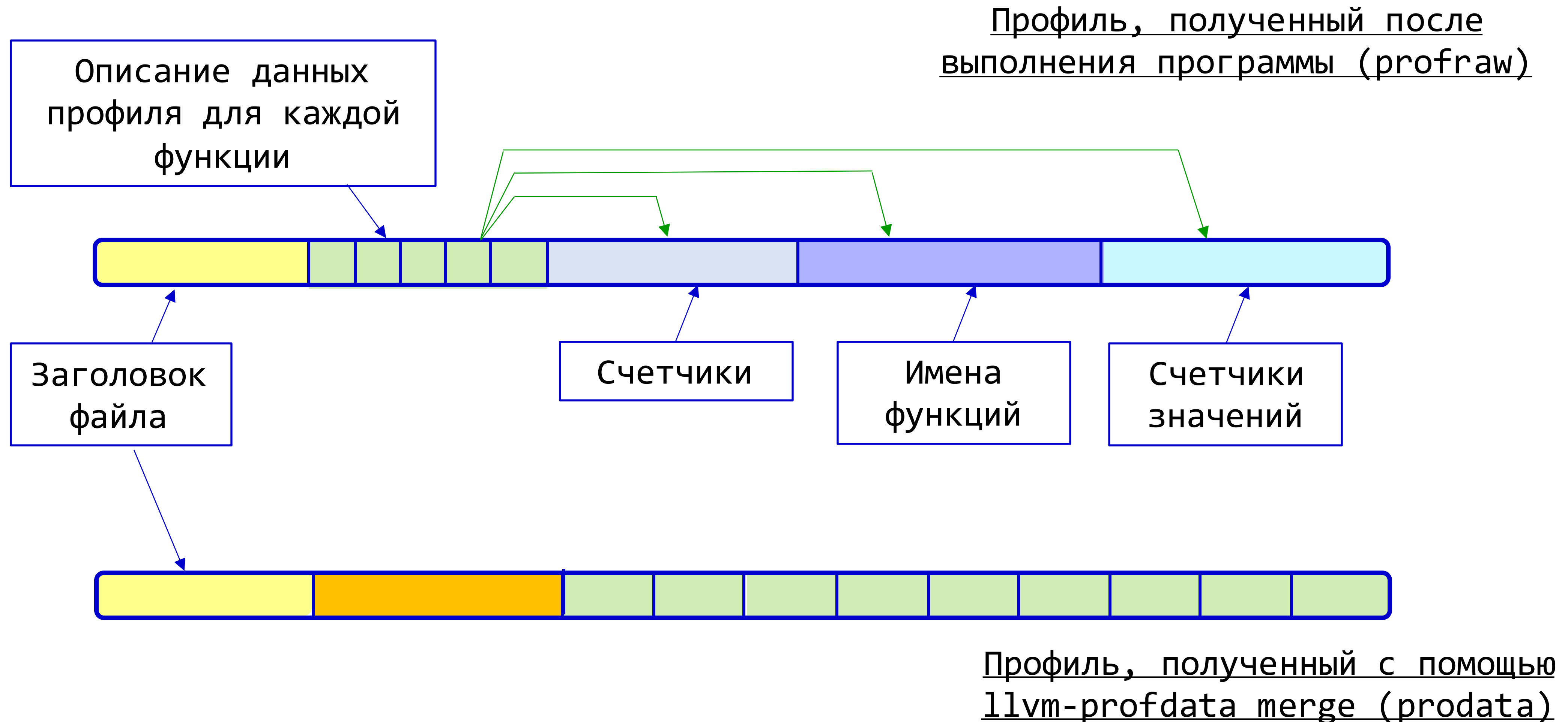
Описание данных профиля для каждой функции

Заголовок файла

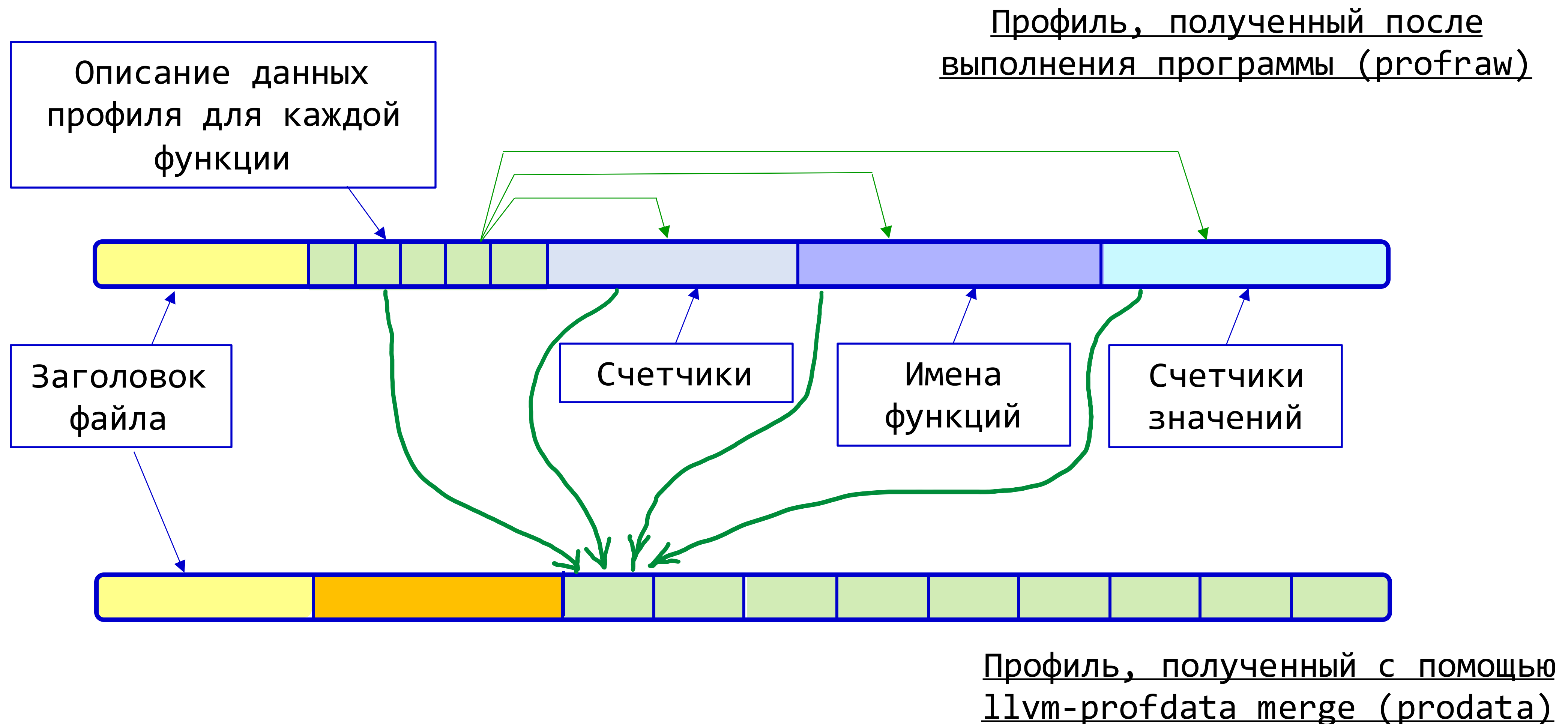
Формат файла с описанием профильной информации



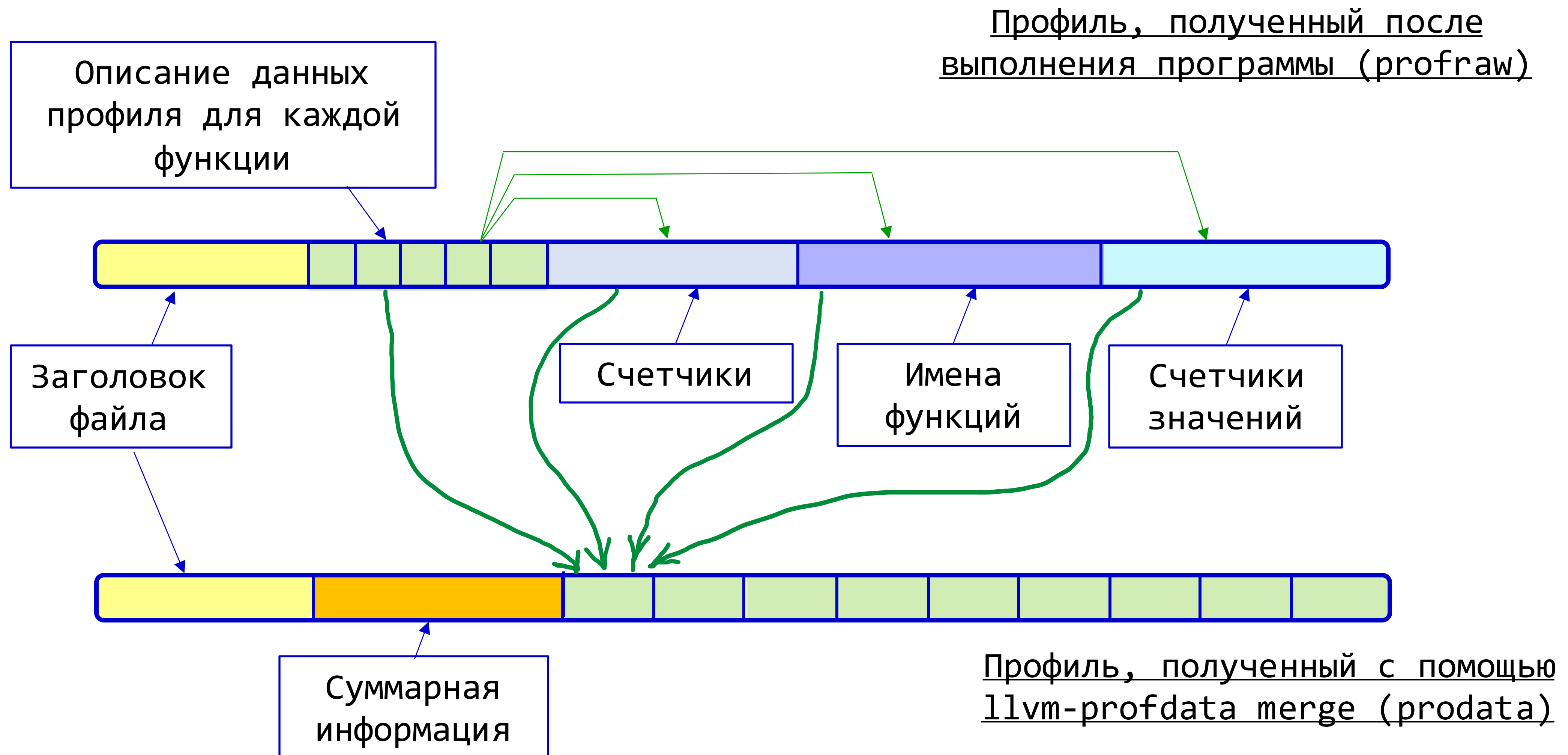
Формат файла с описанием профильной информации



Формат файла с описанием профильной информации



Формат файла с описанием профильной информации



Использование профиля в оптимизациях

- Счетчики выполнения базовых блоков
- Счетчики значений косвенных вызовов
- Счетчики значений запрашиваемых размеров операций с памятью

- Что это
- Как работает
- Как используется для оптимизаций

Вставка счетчиков

```
int bar(int i) {
    if (i > 10) {
        if (i < 100)
            return 42;
        else
            return 777;
    }
    else
        return 0;
}
```

Вставка счетчиков

```
int bar(int i) {
    if (i > 10) {
        if (i < 100)
            return 42;
        else
            return 777;
    }
    else
        return 0;
}
```

```
define i32 @bar(i32 %i) {
entry:
  ...
  br i1 %cmp, label %if.then, label %if.else

if.then:
  ...
  br label %return

if.else:
  ...
  br i1 %cmp1, label %if.then2, label %if.else2

if.then2:
  ...
  br label %return

if.else2:
  ...
  br label %return

return:
  ...
  ret i32 %5
}
```

Вставка счетчиков

```c
int bar(int i) {
    if (i > 10) {
        if (i < 100)
            return 42;
        else
            return 777;
    }
    else
        return 0;
}
```

```llvm
define i32 @bar(i32 %i) {
entry:
  ...
  br i1 %cmp, label %if.then, label %if.else

if.then:
  ...
  br label %return

if.else:
  ...
  br i1 %cmp1, label %if.then2, label %if.else2

if.then2:
  ...
  br label %return

if.else2:
  ...
  br label %return

return:
  ...
  ret i32 %5
}
```

Вставка счетчиков

```
int bar(int i) {
    if (i > 10) {
        if (i < 100)
            return 42;
        else
            return 777;
    }
    else
        return 0;
}
```

```
define i32 @bar(i32 %i) {
entry:
  ...
  br i1 %cmp, label %if.then, label %if.else

if.then:
  ...
  br label %return

if.else:
  ...
  br i1 %cmp1, label %if.then2, label %if.else2

if.then2:
  ...
  br label %return

if.else2:
  ...
  br label %return

return:
  ...
  ret i32 %5
}
```

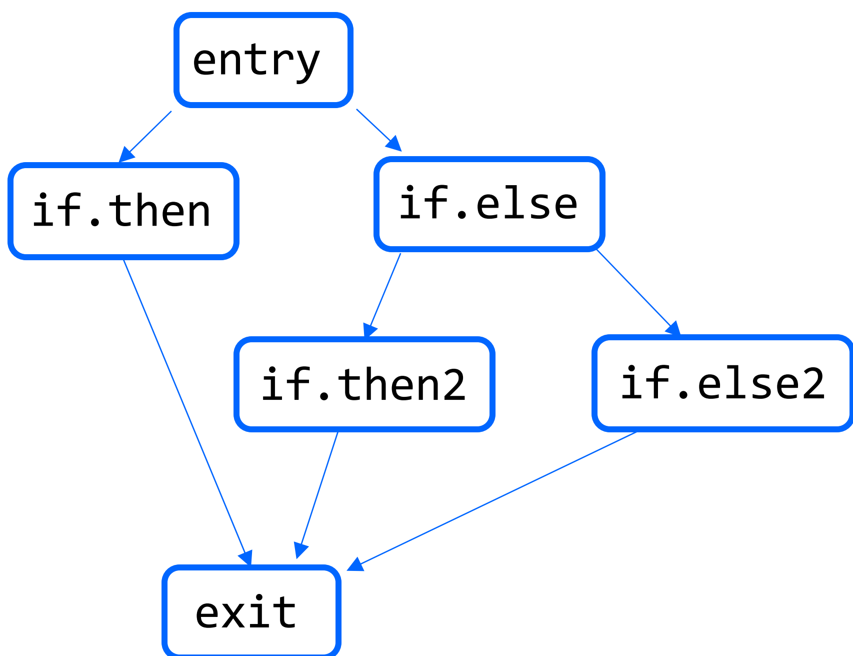
Вставка счетчиков

```
int bar(int i) {
    if (i > 10) {
        if (i < 100)
            return 42;
        else
            return 777;
    }
    else
        return 0;
}
```

```
define i32 @bar(i32 %i) {
entry:
  ...
  br i1 %cmp, label %if.then, label %if.else

if.then:
  ...
  br label %return

if.else:
  ...
  br i1 %cmp1, label %if.then2, label %if.else2

if.then2:
  ...
  br label %return

if.else2:
  ...
  br label %return

return:
  ...
  ret i32 %5
}
```

Вставка счетчиков

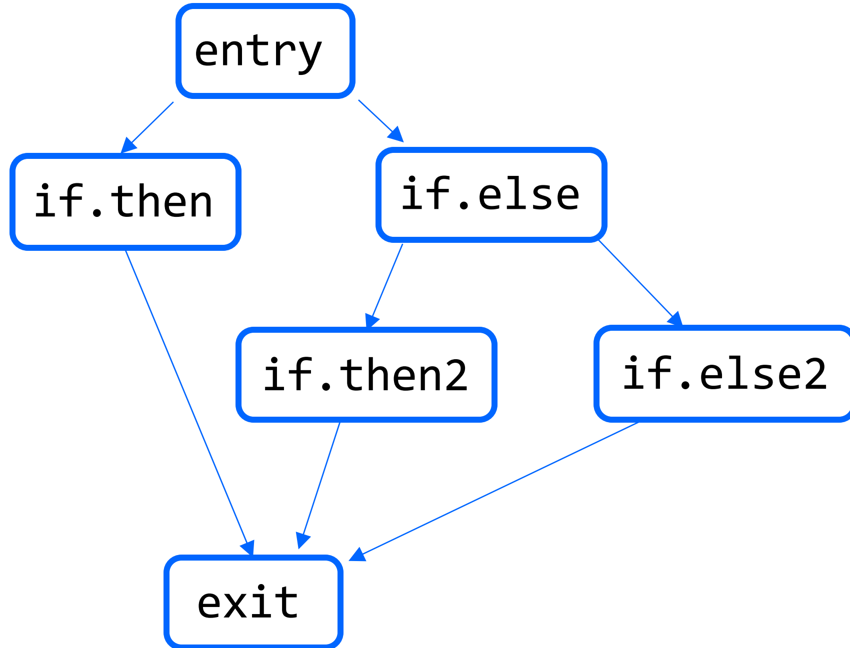
Граф потока управления
Control flow graph (CFG)



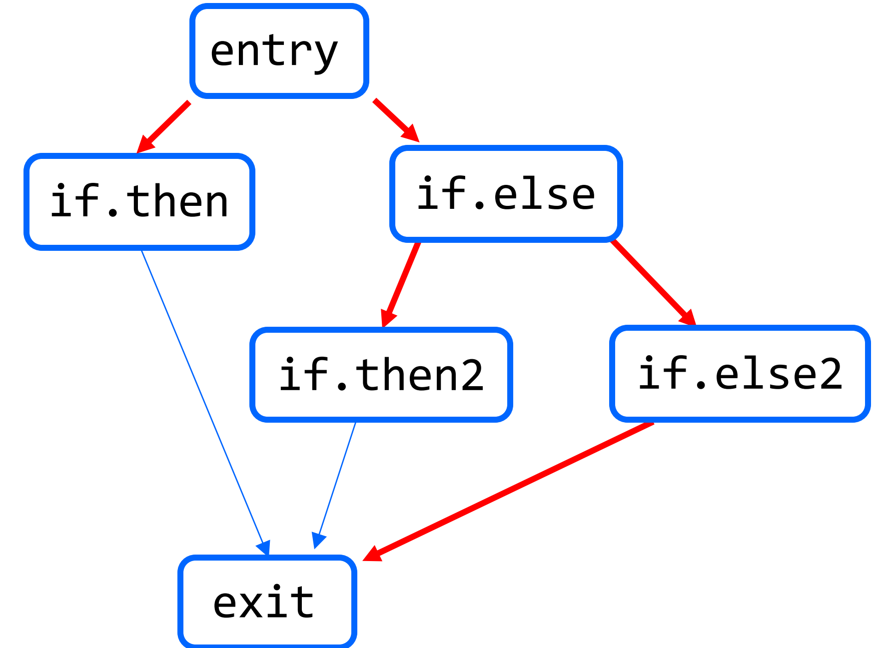

```
define i32 @bar(i32 %i) {
entry:
  ...
  br i1 %cmp, label %if.then, label %if.else

if.then:
  ...
  br label %return

if.else:
  ...
  br i1 %cmp1, label %if.then2, label %if.else2

if.then2:
  ...
  br label %return

if.else2:
  ...
  br label %return

return:
  ...
  ret i32 %5
}
```

Вставка счетчиков

Граф потока управления
Control flow graph (CFG)

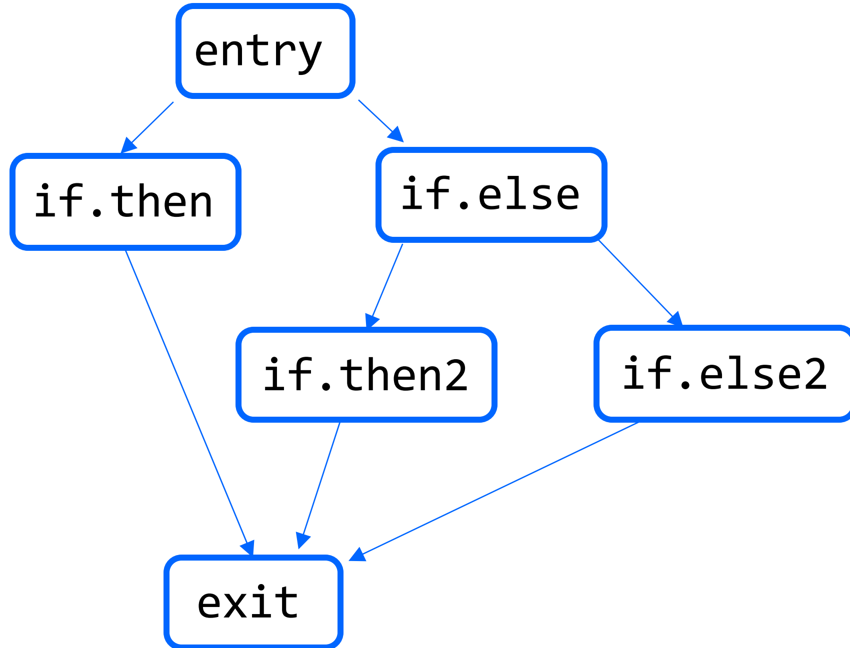


Минимальное остовное дерево
Minimal spanning tree (MST)

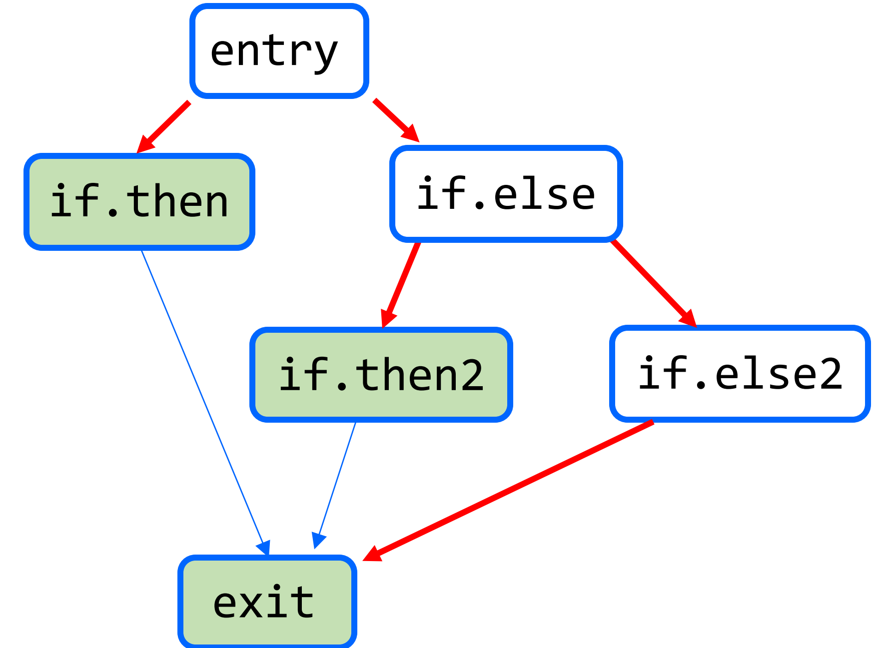


Вставка счетчиков

Граф потока управления
Control flow graph (CFG)

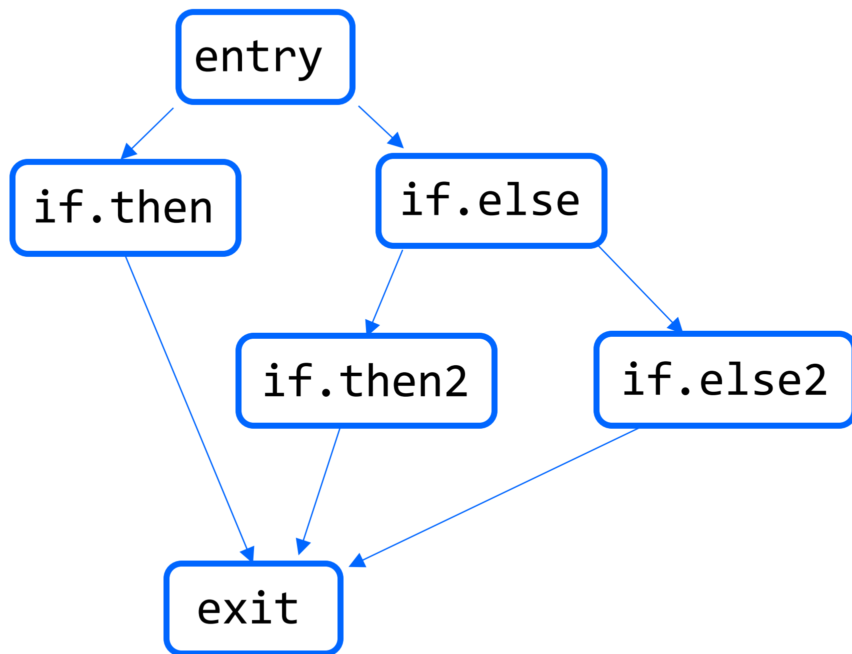


Минимальное остовное дерево
Minimal spanning tree (MST)

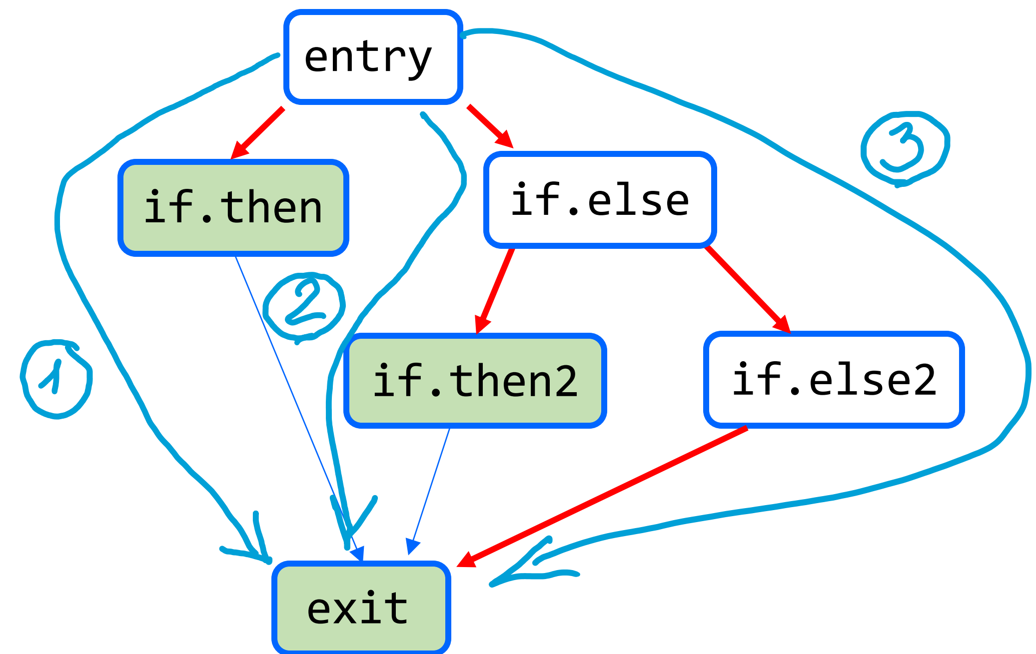


Вставка счетчиков

Граф потока управления
Control flow graph (CFG)

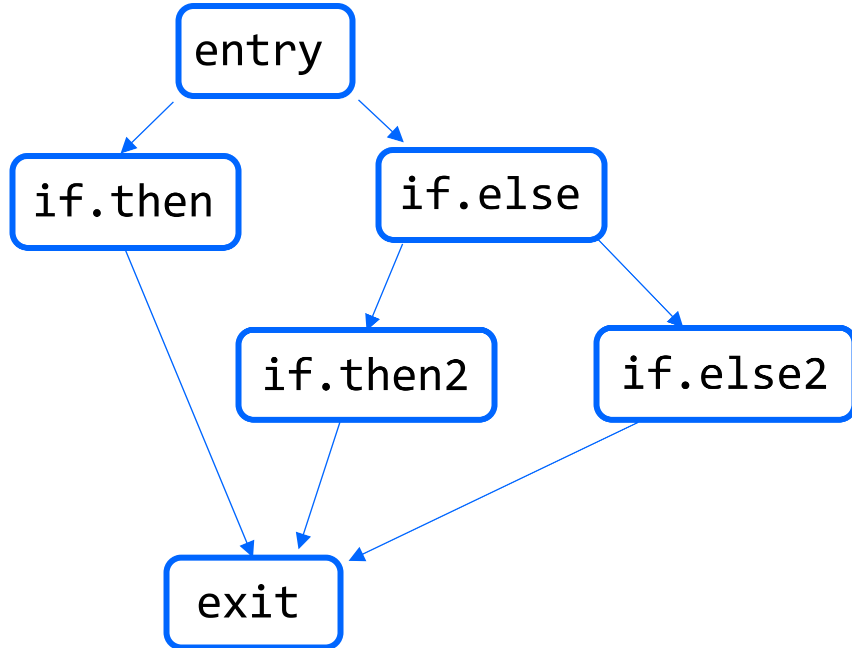


Минимальное остовное дерево
Minimal spanning tree (MST)

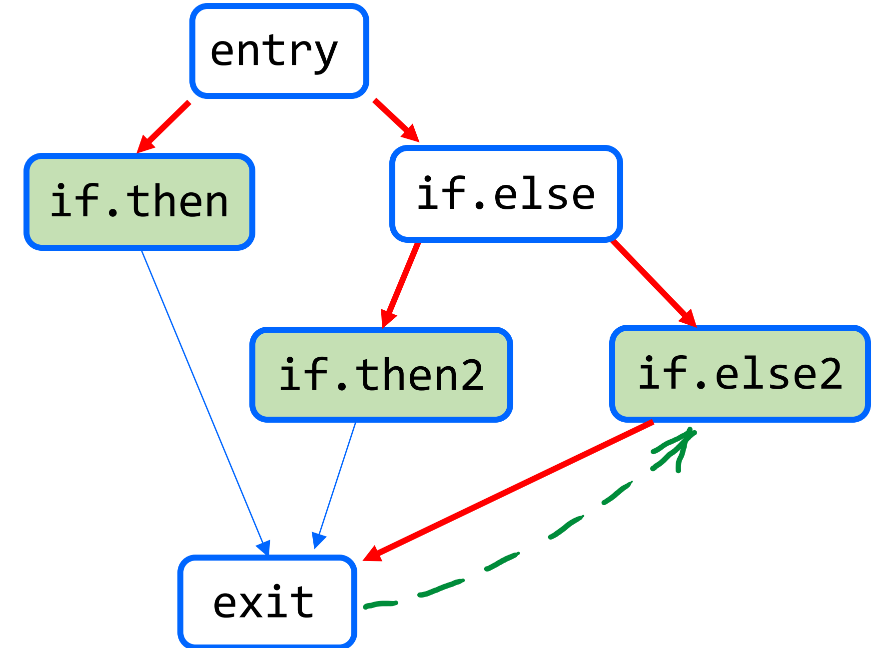


Вставка счетчиков

Граф потока управления
Control flow graph (CFG)

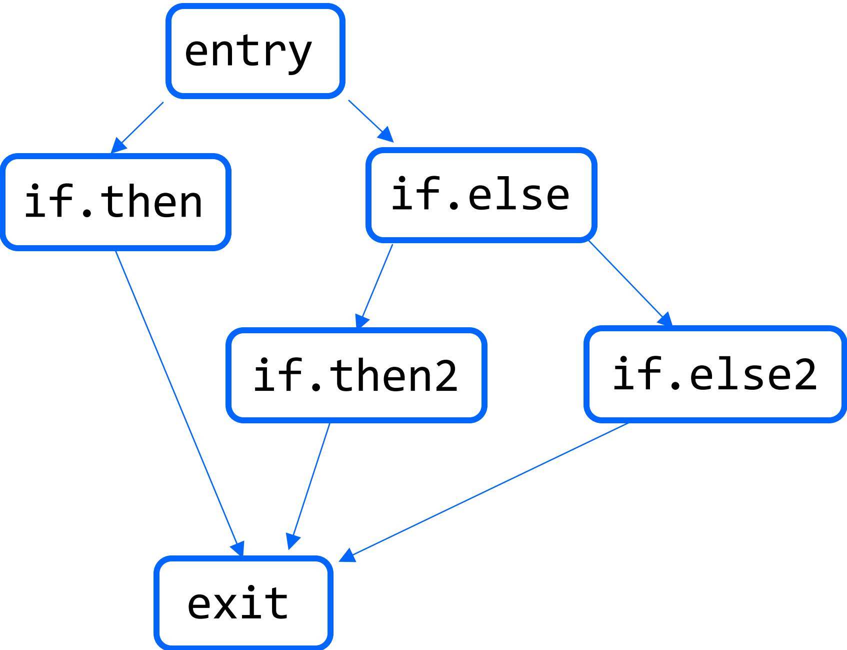


Минимальное остовное дерево
Minimal spanning tree (MST)

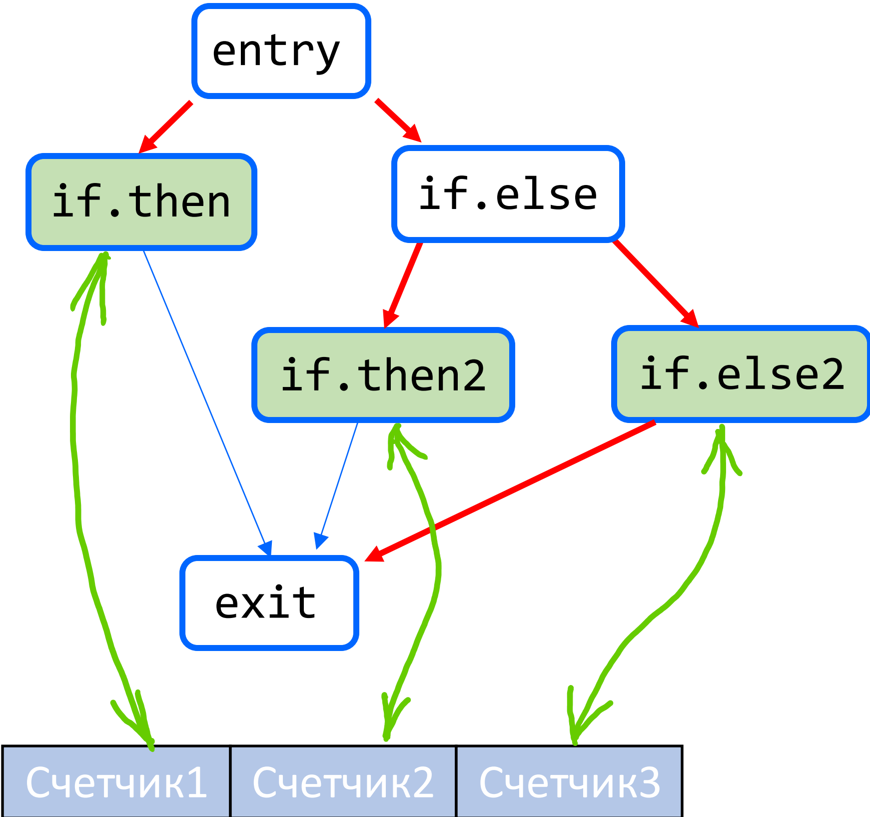


Вставка счетчиков

Граф потока управления
Control flow graph (CFG)

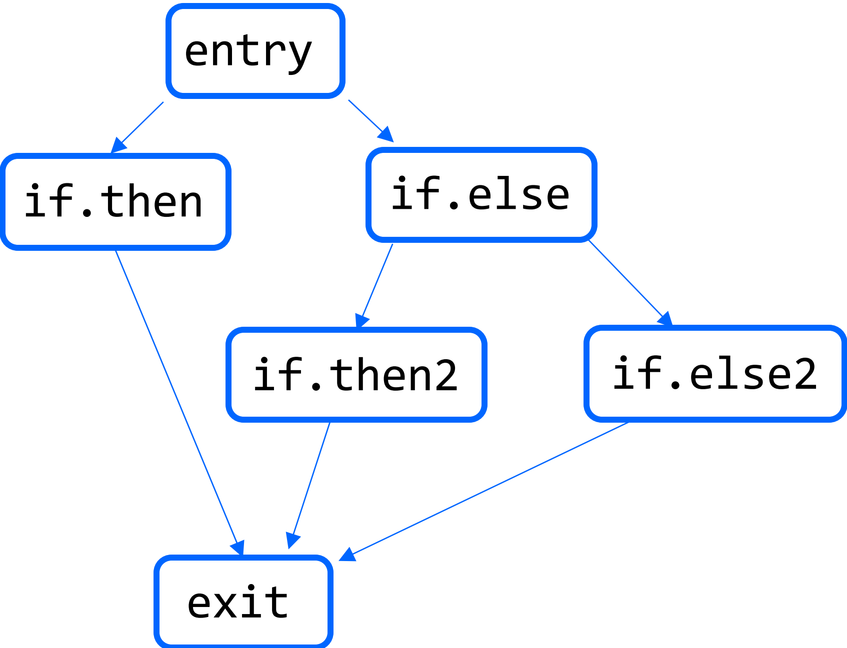


Минимальное остовное дерево
Minimal spanning tree (MST)

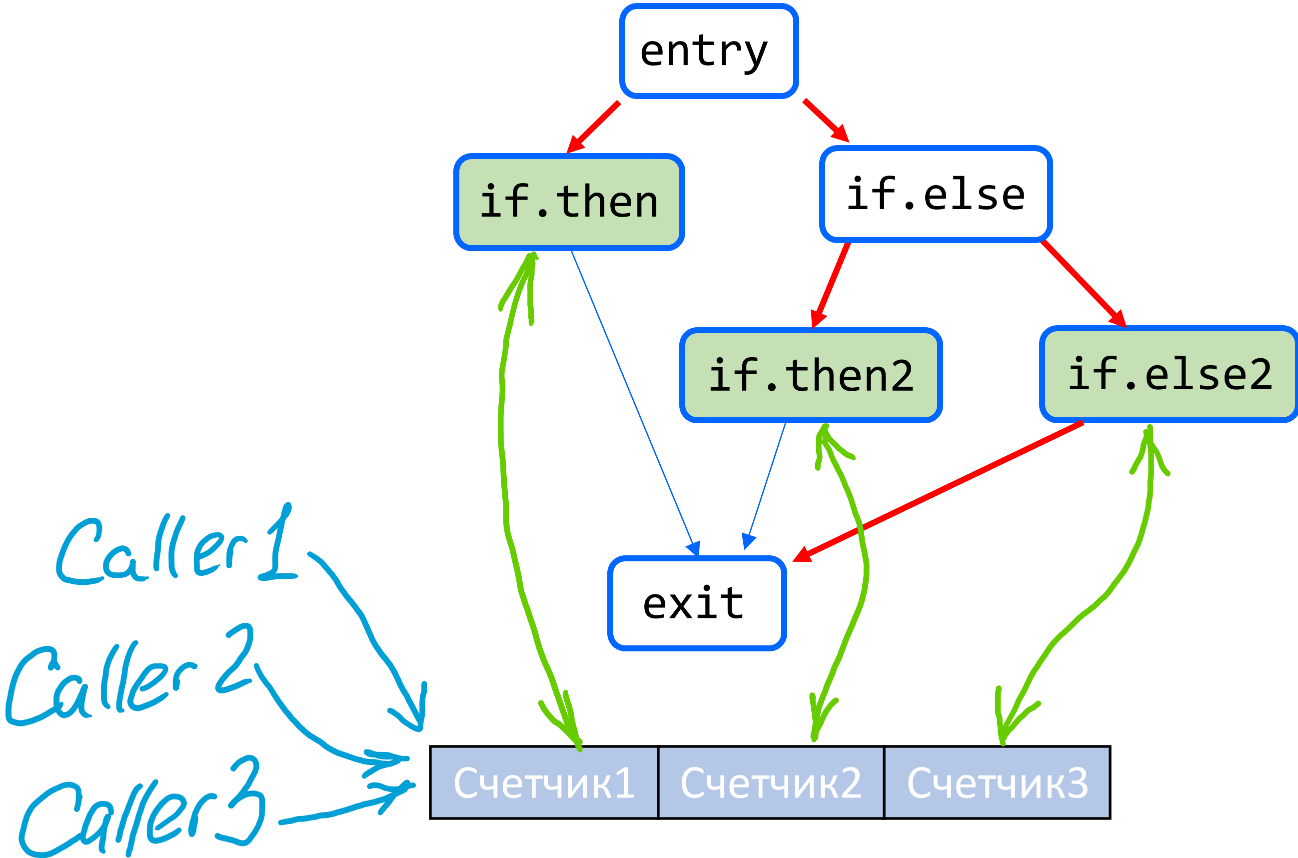


Вставка счетчиков

Граф потока управления
Control flow graph (CFG)



Минимальное остовное дерево
Minimal spanning tree (MST)



Использование счетчиков

```
define i32 @bar(i32 %i) {
entry:
  ...
  br i1 %cmp, label %if.then, label %if.else

if.then:
  ...
  br label %return

if.else:
  ...
  br i1 %cmp1, label %if.then2, label %if.else2

if.then2:
  ...
  br label %return

if.else2:
  ...
  br label %return

return:
  ...
  ret i32 %5
}
```

Использование счетчиков

```
define i32 @bar(i32 %i) {
entry:
  ...
  br i1 %cmp, label %if.then, label %if.else

if.then:
  ...
  br label %return

if.else:
  ...
  br i1 %cmp1, label %if.then2, label %if.else2

if.then2:
  ...
  br label %return

if.else2:
  ...
  br label %return

return:
  ...
  ret i32 %5
}
```

Использование счетчиков

```
define i32 @bar(i32 %i) {
entry:
  ...
  br i1 %cmp, label %if.then, label %if.else

if.then:
  ...
  br label %return

if.else:
  ...
  br i1 %cmp1, label %if.then2, label %if.else2

if.then2:
  ...
  br label %return

if.else2:
  ...
  br label %return

return:
  ...
  ret i32 %5
}
```

```
define i32 @bar(i32 %i) {
entry:
  ...
  br i1 %cmp, label %if.then, label %if.else

if.else:
  ...
  br i1 %cmp1, label %if.then2, label %if.else2

if.else2:
  ...
  br label %return

return:
  ...
  ret i32 %5

if.then2:
  ...
  br label %return

if.then:
  ...
  br label %return
}
```

Использование счетчиков (встраивание функций)

```
define i32 @bar(i32 %i) {
entry:
  ...
  br i1 %cmp, label %if.then, label %if.else

if.then:
  ...
  br label %return

if.else:
  ...
  br i1 %cmp1, label %if.then2, label %if.else2

if.then2:
  ...
  br label %return

if.else2:
  ...
  br label %return

return:
  ...
  ret i32 %5
}
```

Использование счетчиков (встраивание функций)

```
define i32 @bar(i32 %i) {
entry:
  ...
  br i1 %cmp, label %if.then, label %if.else

if.then:
  ...
  br label %return

if.else:
  ...
  br i1 %cmp1, label %if.then2, label %if.else2

if.then2:
  ...
  br label %return

if.else2:
  ...
  br label %return

return:
  ...
  ret i32 %5
}
```

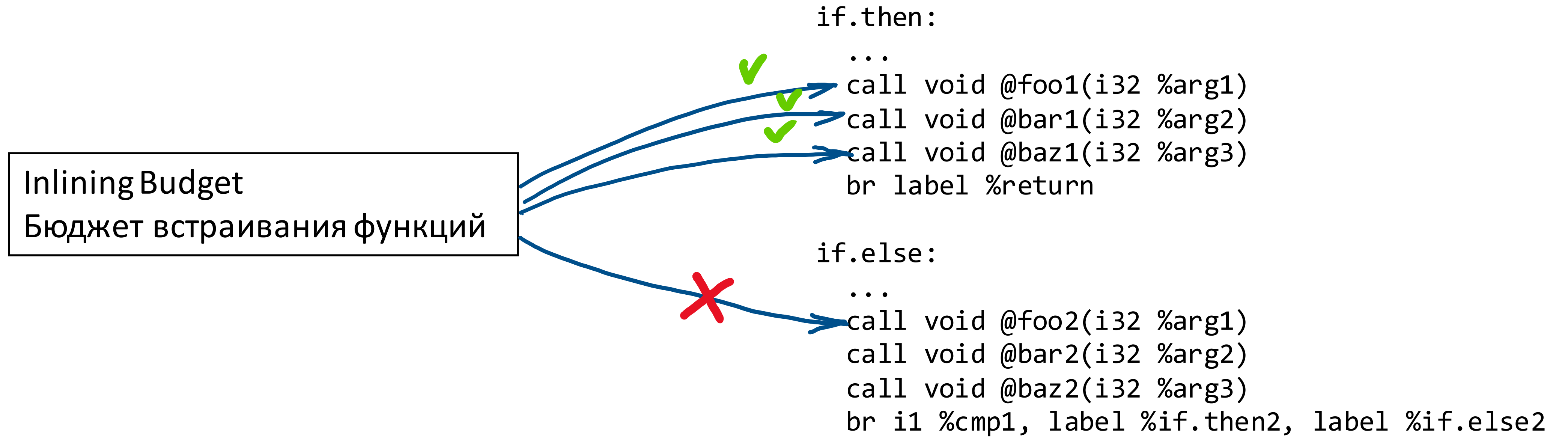
```
if.then:
  ...
  call void @foo1(i32 %arg1)
  call void @bar1(i32 %arg2)
  call void @baz1(i32 %arg3)
  br label %return

if.else:
  ...
  call void @foo2(i32 %arg1)
  call void @bar2(i32 %arg2)
  call void @baz2(i32 %arg3)
  br i1 %cmp1, label %if.then2, label %if.else2
```

Использование счетчиков (встраивание функций)

Inlining Budget
Бюджет встраивания функций

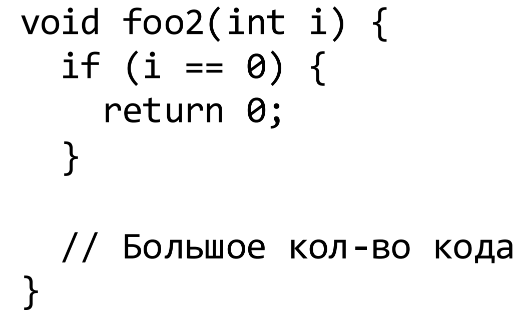
```
if.then:
  ...
  call void @foo1(i32 %arg1)
  call void @bar1(i32 %arg2)
  call void @baz1(i32 %arg3)
  br label %return

if.else:
  ...
  call void @foo2(i32 %arg1)
  call void @bar2(i32 %arg2)
  call void @baz2(i32 %arg3)
  br i1 %cmp1, label %if.then2, label %if.else2
```

Использование счетчиков (встраивание функций)



Использование счетчиков (частичное встраивание функций)

```
if.else:
  ...
  call void @foo2(i32 %arg1)
  call void @bar2(i32 %arg2)
  call void @baz2(i32 %arg3)
  br i1 %cmp1, label %if.then2, label %if.else2
```



```
void foo2(int i) {
  if (i == 0) {
    return 0;
  }

  // Большое кол-во кода
}
```

Профилирование значений. Косвенные вызовы (indirect calls)

1. При инструментации происходит вставка кода, обеспечивающего последующую запись адреса функции

```
call void @__llvm_profile_instrument_target(i64 %funcptr, i8* @__profd_main, i32 0)
```

Профилирование значений. Косвенные вызовы (indirect calls)

1. При инструментации происходит вставка кода, обеспечивающего последующую запись адреса функции
2. Запись адреса в профиль на стадии линковки

Профилирование значений. Косвенные вызовы (indirect calls)

1. При инструментации происходит вставка кода, обеспечивающего последующую запись адреса функции
2. Запись адреса в профиль на стадии линковки
3. Запись значений адресов и количеств вызовов во время выполнения программы
    - По умолчанию - 16, можно переопределить с помощью переменной окружения LLVM_VP_MAX_NUM_VALS_PER_SITE , но не получится сделать больше, чем INSTR_PROF_MAX_NUM_VAL_PER_SITE (255)
    - Запись значений в формате адрес и количество вызовов

Профилирование значений. Косвенные вызовы (indirect calls)

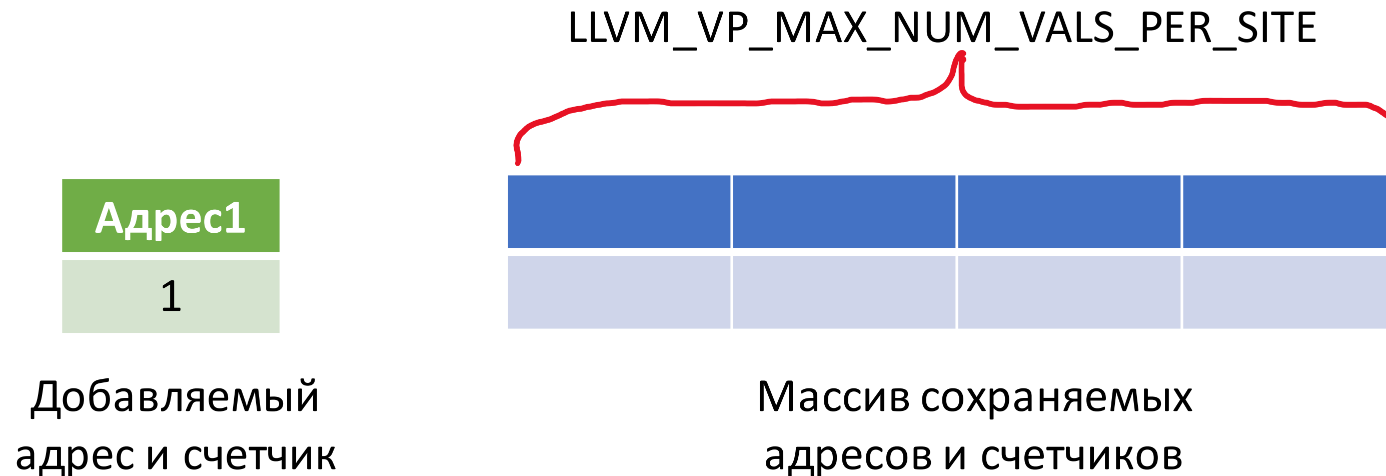
1. При инструментации происходит вставка кода, обеспечивающего последующую запись адреса функции
2. Запись адреса в профиль на стадии линковки
3. Запись значений адресов и количеств вызовов во время выполнения программы
   - По умолчанию - 16, можно переопределить с помощью переменной окружения LLVM_VP_MAX_NUM_VALS_PER_SITE , но не получится сделать больше, чем INSTR_PROF_MAX_NUM_VAL_PER_SITE (255)
   - Запись значений в формате адрес и количество вызовов
4. Сопоставление адреса и имени функции при постобработке профиля

Профилирование значений. Косвенные вызовы (indirect calls)

1. При инструментации происходит вставка кода, обеспечивающего последующую запись адреса функции
2. Запись адреса в профиль на стадии линковки
3. Запись значений адресов и количеств вызовов во время выполнения программы
   - По умолчанию - 16, можно переопределить с помощью переменной окружения LLVM_VP_MAX_NUM_VALS_PER_SITE , но не получится сделать больше, чем INSTR_PROF_MAX_NUM_VAL_PER_SITE (255)
   - Запись значений в формате адрес и количество вызовов
4. Сопоставление адреса и имени функции при постобработке профиля
5. Сопоставление имени функции и объекта функции в промежуточном представлении на стадии подгрузки профиля в компилятор

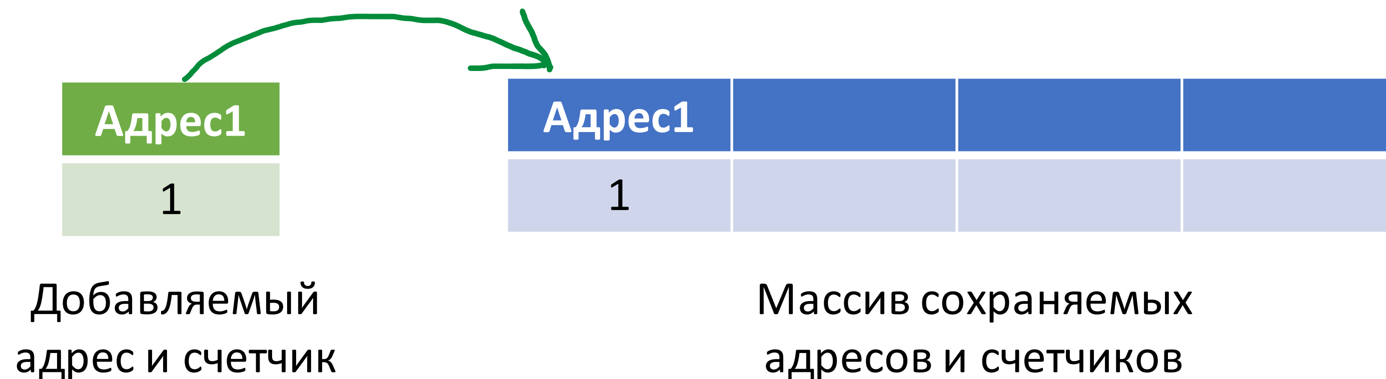
Профилирование значений. Косвенные вызовы (indirect calls)

Алгоритм добавления профилируемых значений



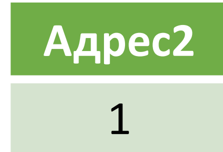
Профилирование значений. Косвенные вызовы (indirect calls)

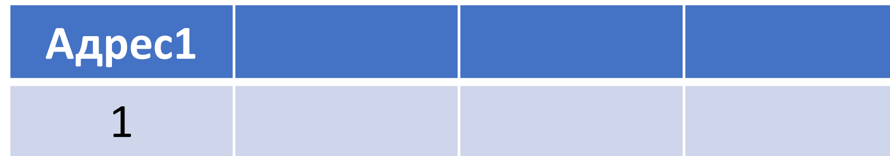
Алгоритм добавления профилируемых значений



Профилирование значений. Косвенные вызовы (indirect calls)

Алгоритм добавления профилируемых значений



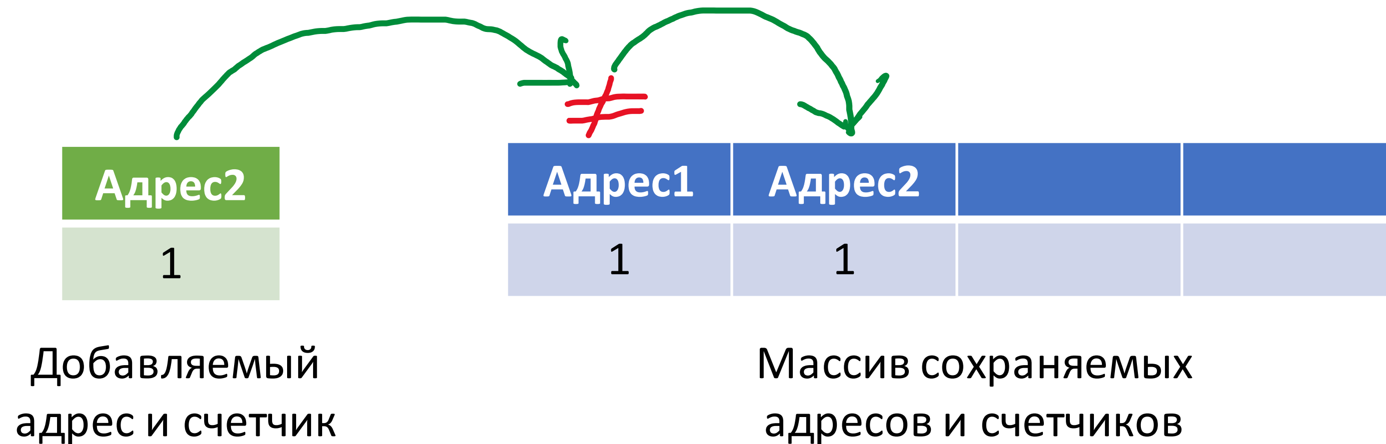


Добавляемый адрес и счетчик

Массив сохраняемых адресов и счетчиков

Профилирование значений. Косвенные вызовы (indirect calls)

Алгоритм добавления профилируемых значений



Добавляемый адрес и счетчик

Массив сохраняемых адресов и счетчиков

Профилирование значений. Косвенные вызовы (indirect calls)

Алгоритм добавления профилируемых значений

Адрес5
1

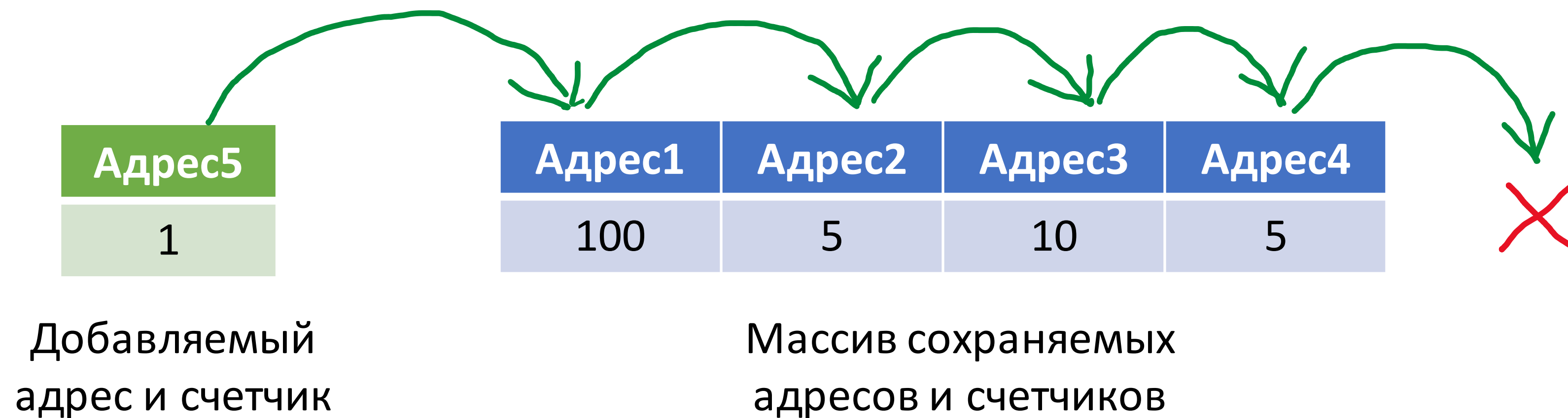
Добавляемый адрес и счетчик

Адрес1	Адрес2	Адрес3	Адрес4
100	5	10	5

Массив сохраняемых адресов и счетчиков

Профилирование значений. Косвенные вызовы (indirect calls)

Алгоритм добавления профилируемых значений

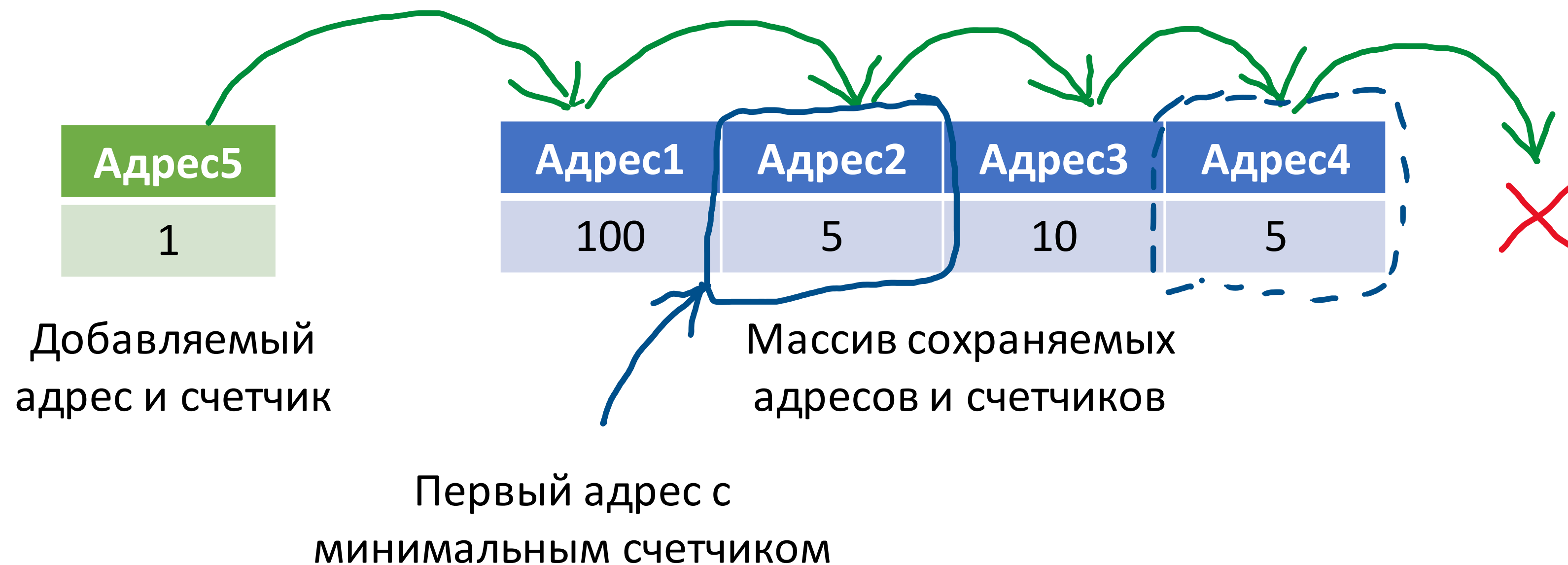


Добавляемый адрес и счетчик

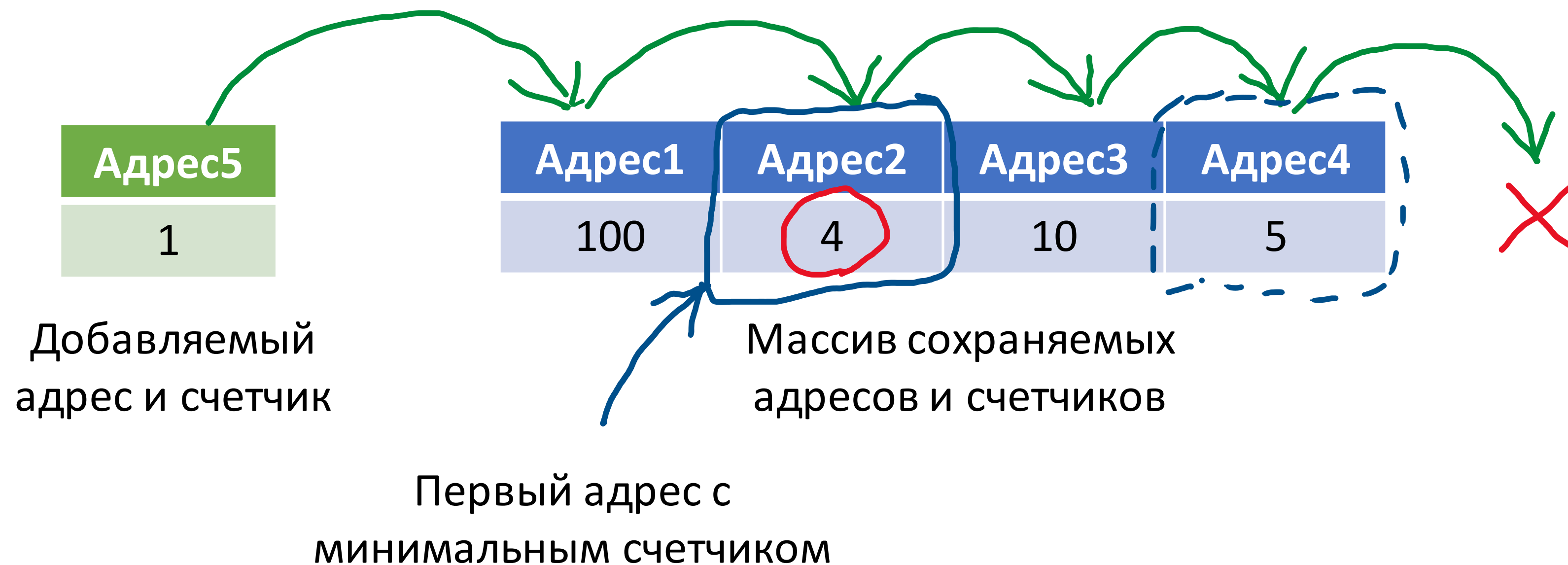
Массив сохраняемых адресов и счетчиков

Профилирование значений. Косвенные вызовы (indirect calls)

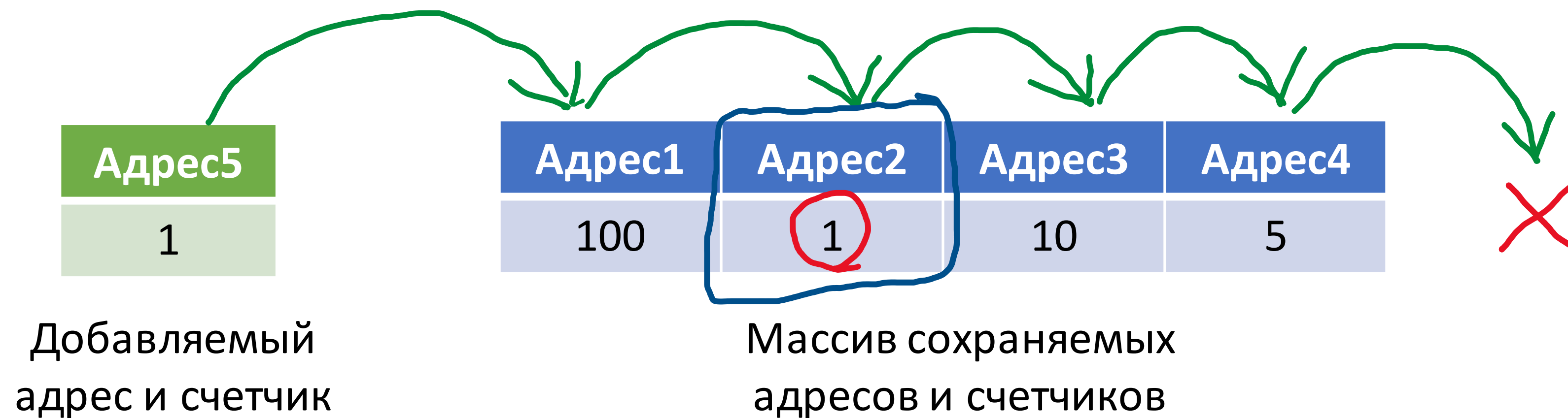
Алгоритм добавления профилируемых значений



Профилирование значений. Косвенные вызовы (indirect calls)

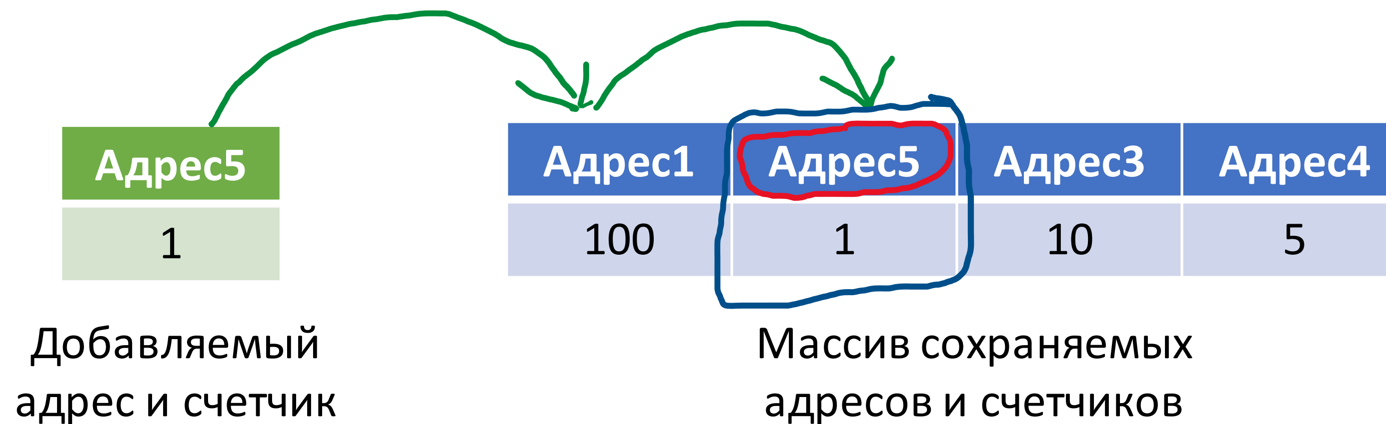


Профилирование значений. Косвенные вызовы (indirect calls)



Профилирование значений. Косвенные вызовы (indirect calls)

Алгоритм добавления профилируемых значений



Добавляемый адрес и счетчик

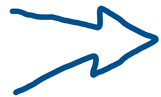
Массив сохраняемых адресов и счетчиков

Использование полученных значений адресов косвенных вызовов

```
using funcptr = void(*)();

void foo(int index) {
  ...
  funcptr f = getfunc(index);
  f();
  ...
}
```

Использование полученных значений адресов косвенных вызовов

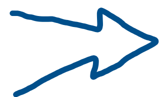
```
using funcptr = void(*)();

void foo(int index) {
  ...
  funcptr f = getfunc(index);
  f();
  ...
}
```



```
...
callq   getfunc
callq   *%rax
...
```

Использование полученных значений адресов косвенных вызовов

```
using funcptr = void(*)();

void foo(int index) {
  ...
  funcptr f = getfunc(index);
  f();
  ...
}
```



```
...
callq   getfunc
callq   *%rax
...
```

```
    ...
    movl    $bar, %r12
    callq   getfunc
    cmpq    %r12, %rax
    jne .false_cmp
    callq   bar
    ...
.false_cmp:
    callq *%rax
    ...
```

Профилирование значений. Размеры операций с памятью (memmove, memcpy, memset)

1. При инструментации происходит вставка кода, обеспечивающего запись запрашиваемого размера

```
void * memmove ( void * dst, const void * src, size_t num );
void * memcpy  ( void * dst, const void * src, size_t num );
void * memset  ( void * ptr, int value,        size_t num );
```

Профилирование значений. Размеры операций с памятью (memmove, memcpy, memset)

1. При инструментации происходит вставка кода, обеспечивающего запись запрашиваемого размера

```
void * memmove ( void * dst, const void * src, size_t num );
void * memcpy  ( void * dst, const void * src, size_t num );
void * memset  ( void * ptr, int value,        size_t num );
```

```
// Исходный код
memset(ptr, 0, size)

// LLVM-IR
call void @__llvm_profile_instrument_range(i64 %size, i8* @__profd_main1, i32 0, i64 8, i64 256, i64 8192)
call void @llvm.memset.p0i8.i64(i8* align 8 %1, i8 0, i64 %size, i1 false)
```

Профилирование значений. Размеры операций с памятью (memmove, memcpy, memset)

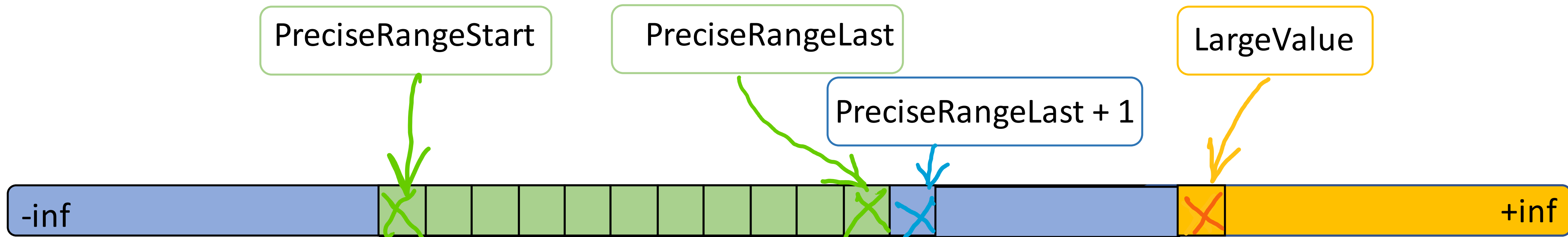
1. При инструментации происходит вставка кода, обеспечивающего запись запрашиваемого размера

```
void * memmove ( void * dst, const void * src, size_t num );
void * memcpy  ( void * dst, const void * src, size_t num );
void * memset  ( void * ptr, int value,        size_t num );
```

2. Во время выполнения данные об этих размерах собираются
3. Передаются в компилятор для применения оптимизации

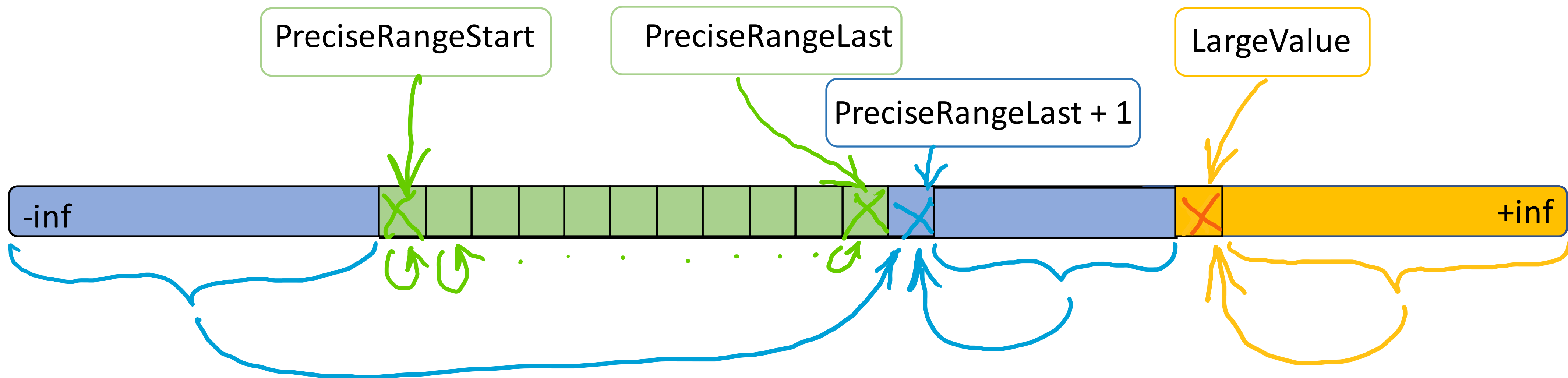
Профилирование значений. Размеры операций с памятью (memmove, memcpy, memset)

Алгоритм записи значений



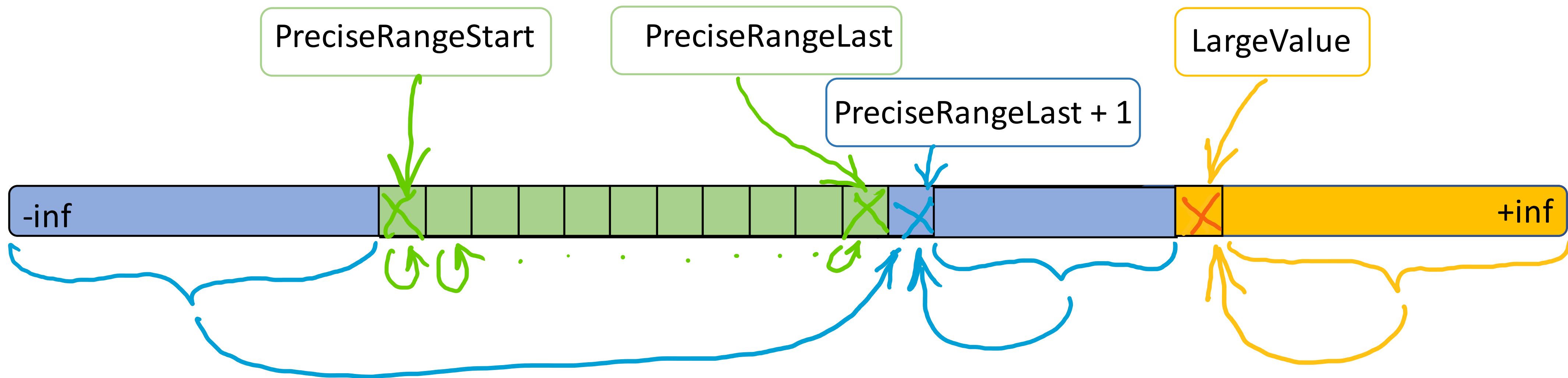
Профилирование значений. Размеры операций с памятью (memmove, memcpy, memset)

Алгоритм записи значений



Профилирование значений. Размеры операций с памятью (memmove, memcpy, memset)

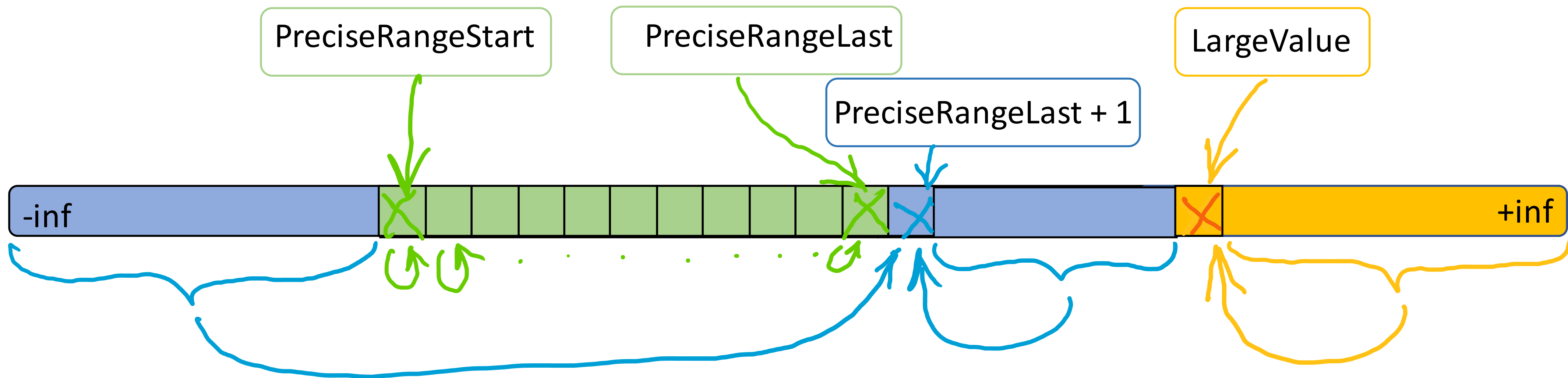
Алгоритм записи значений



-mllvm -memop-size-range=32:64
-mllvm -memop-size-large=8192

Профилирование значений. Размеры операций с памятью (memmove, memcpy, memset)

Алгоритм записи значений



-mllvm -memop-size-range=32:64
-mllvm -memop-size-large=8192

LLVM_VP_MAX_NUM_VALS_PER_SITE
INSTR_PROF_MAX_NUM_VAL_PER_SITE (255)

Использование размеров операций с памятью для оптимизации

```
memset(ptr, 0, some_val)
```

```
switch (some_val) {
  case 4: memset(ptr, 0, 4); break;
  case 13: memset(ptr, 0, 13); break;
  case 32: memset(ptr, 0, 32); break;
  default: memset(ptr, 0, some_val); break;
}
```

```
movl   $0, 16(%rsp)
```

```
callq memset
```

Семплирование

1. Сборка с отладочной информацией (достаточно таблицы строк исходного файла)
   ```
   > clang++ -O2 -gline-tables-only code.cc -o code
   ```
2. Запуск программы с помощью семплирующего профилировщика
   ```
   > perf record -b ./code
   ```
3. Преобразование формата собранного профиля
   ```
   > create_llvm_prof --binary=./code --out=code.prof
   ```
4. Сборка с использованием профиля
   ```
   > clang++ -O2 -gline-tables-only \
       -fprofile-sample-use=code.prof code.cc -o code
   ```

Сэмплирование. Детали

- Принцип работы perf:
  - Прерывание выполнения запущенной программы
  - Получение текущей инструкции программы и увеличение ее счетчика
  - Запись значений аппаратных счетчиков (PMU) для определения количества циклов, инструкций и т.д.
  - Запись значений программных счетчиков (ядро операционной системы) для определения количества переключений контекста, TLB-промахов, и т.д.
- create_llvm_prof:
  - Инструмент из проекта https://github.com/google/autofdo

Инструментация VS Сэмплирование

- Инструментация:
  - Детерминированность
  - Точность
  - Больше информации для оптимизаций (косвенные вызовы, значения запрашиваемых размеров в операциях с памятью)
- Сэмплирование:
  - Меньший оверхед
  - Можно использовать на оптимизированных бинарных файлах

Инструментация с учетом контекста. Context-sensitive PGO

- Проблемы инструментации: выполняется до инлайнинга
- Искажение значений счетчиков из-за встраивания функций
- Может привести к ухудшению даже основного сценария

Инструментация с учетом контекста. Context-sensitive PGO (CS-PGO)

1. Сборка инструментированной версии
```
> clang++ -O2 -fprofile-generate code.cc -o code
```
2. Запуск инструментированной версии
```
> LLVM_PROFILE_FILE="code-%p.profraw" ./code
```
3. Объединение профилей и конвертирование их в формат, ожидаемый компилятором
```
> llvm-profdata merge -output=code.profdata code-*.profraw
```
4. Сборка версии, оптимизированной с использованием профиля
```
> clang++ -O2 -fprofile-use=code.profdata code.cc -o code
```

LLVM IR PGO

1. Сборка инструментированной версии с учетом контекста
```
> clang++ -O2 -fprofile-use=code.profdata -fcs-profile-generate -o cs_code
```
2. Запуск инструментированной версии
```
> ./cs_code
```
3. Объединение профилей и конвертирование их в формат, ожидаемый компилятором
```
> llvm-profdata merge -output=cs_code.profdata code.profdata
```
4. Сборка версии, оптимизированной с использованием профиля
```
> clang++ -O2 -fprofile-use=cs_code.profdata
```

LLVM IR CS PGO

Оптимизация бинарных файлов. BOLT

- Binary Optimizer and Layout Tool
- Бинарный файл с отладочной информацией (unstripped)
- AST -> LLVM IR -> MachineInstr -> MCInstr
- Более 10 оптимизаций
  - Свертка одинаковых участков кода (identical code folding)
  - Девиртуализация вызовов (indirect call promotion)
  - Переупорядочивание базовых блоков (reorder basic blocks)
  - Оптимизация раскладки кода (layout optimization: split hot/cold blocks)
  - Удаление ненужных сохранений регистров на стеке при вызове функций (removes unnecessary caller-saved register spilling)
  - ...

Итоги

- **Плюс PGO:** выбранный сценарий программы может довольно серьезно улучшиться
- **Минусы PGO:** остальные сценарии применения программы могут просесть в производительности; сбор профиля программы может существенно осложнить систему сборки приложения
- **Есть, что улучшать:** использование профилей в большем количестве оптимизаций; привязка счетчиков к местам вызова функций; уменьшение накладных расходов от инструментации; улучшение точности PGO, основанного на семплировании (напр: Context-Sensitive SamplePGO)

Заметки

- PGO вряд ли сможет помочь, если на горячем участке ассемблерные вставки, компилятор их не оптимизирует. Но можно попробовать применить BOLT.
- Инструментация для получения графа вызова функций с возможностью отключения в рантайме: `-fxray-instrument` (https://llvm.org/docs/XRay.html)
- Для профилирования динамических библиотек можно вручную вызывать

```
__llvm_profile_reset_counters();
... горячий участок кода 1
__llvm_profile_dump();
... код, который не нуждается в профилировании
__llvm_profile_reset_counters();
... горячий участок кода 2
__llvm_profile_dump();
```

Спасибо!