Задачи анализа и оптимизации программ

6 марта 2019 года

Андрей Белеванцев <abel@ispras.ru>

Отдел компиляторных технологий

Институт системного программирования им. В.П. Иванникова РАН

О чем эта презентация

- Значение компиляторных технологий
- Как устроен оптимизирующий компилятор
- Некоторые работы ИСП РАН
  - Компиляторы GCC и LLVM
  - JIT-компиляция
  - Статический анализ исходного кода
- Анализ бинарного кода: Вартан Падарян, 13 марта

Вызовы в IT и компиляторы

- Безопасность
  - Все приложения в глобальной сети, с открытым кодом
  - Как защититься от атак, предотвратить атаки, обеспечить целостность данных?

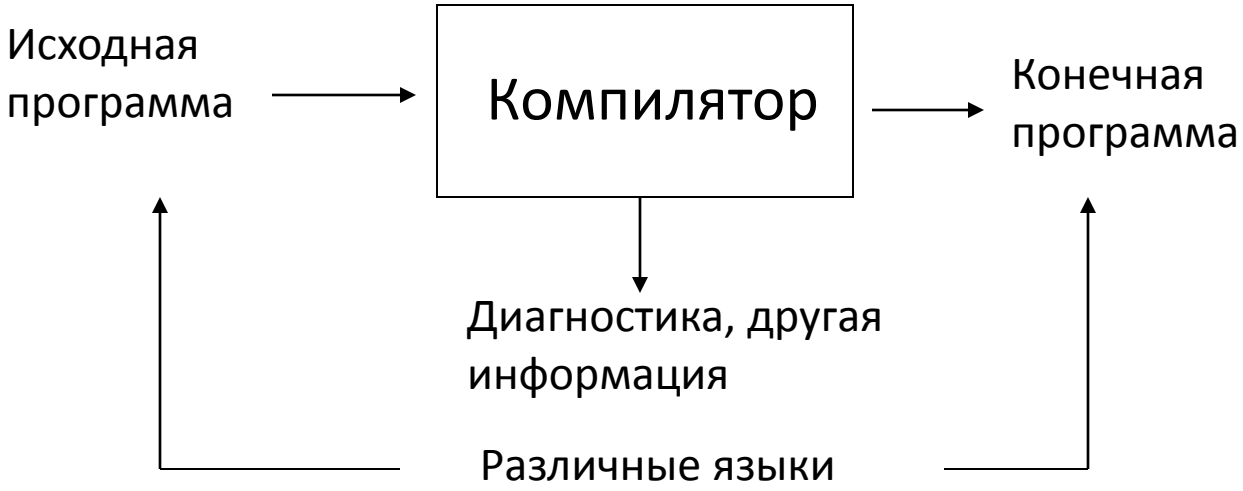
  →Инструменты для поиска уязвимостей и критических ошибок (статические/динамические, исходный/бинарный код, разные языки программирования)
- Параллелизм на всех уровнях аппаратуры
  - Внутри процессора, внутри узла распределенной системы, между узлами
  - Как программировать?

  →Инструменты для выражения программ на языках высокого уровня на языке аппаратуры (на одном процессоре, многоядерные/гетерогенные/распределенные системы, статические/динамические языки)

Компиляторные технологии

- Оптимизация программ (исходный код)
  - Статическая (традиционные компиляторы)
  - Динамическая (Java, JavaScript, …)
- Двоичная трансляция и оптимизация
  - Симуляторы (QEMU)
  - Динамические инструменты анализа (Valgrind …)
- Статический анализ программ
  - Критические ошибки, уязвимости (Coverity)
  - Понимание программ (Understand)
- Динамический анализ программ
  - Поиск ошибок (фаззинг, символьное выполнение)
  - Отладка (backward debugging)
  - Профилирование: производительность, память, …

Компиляторы

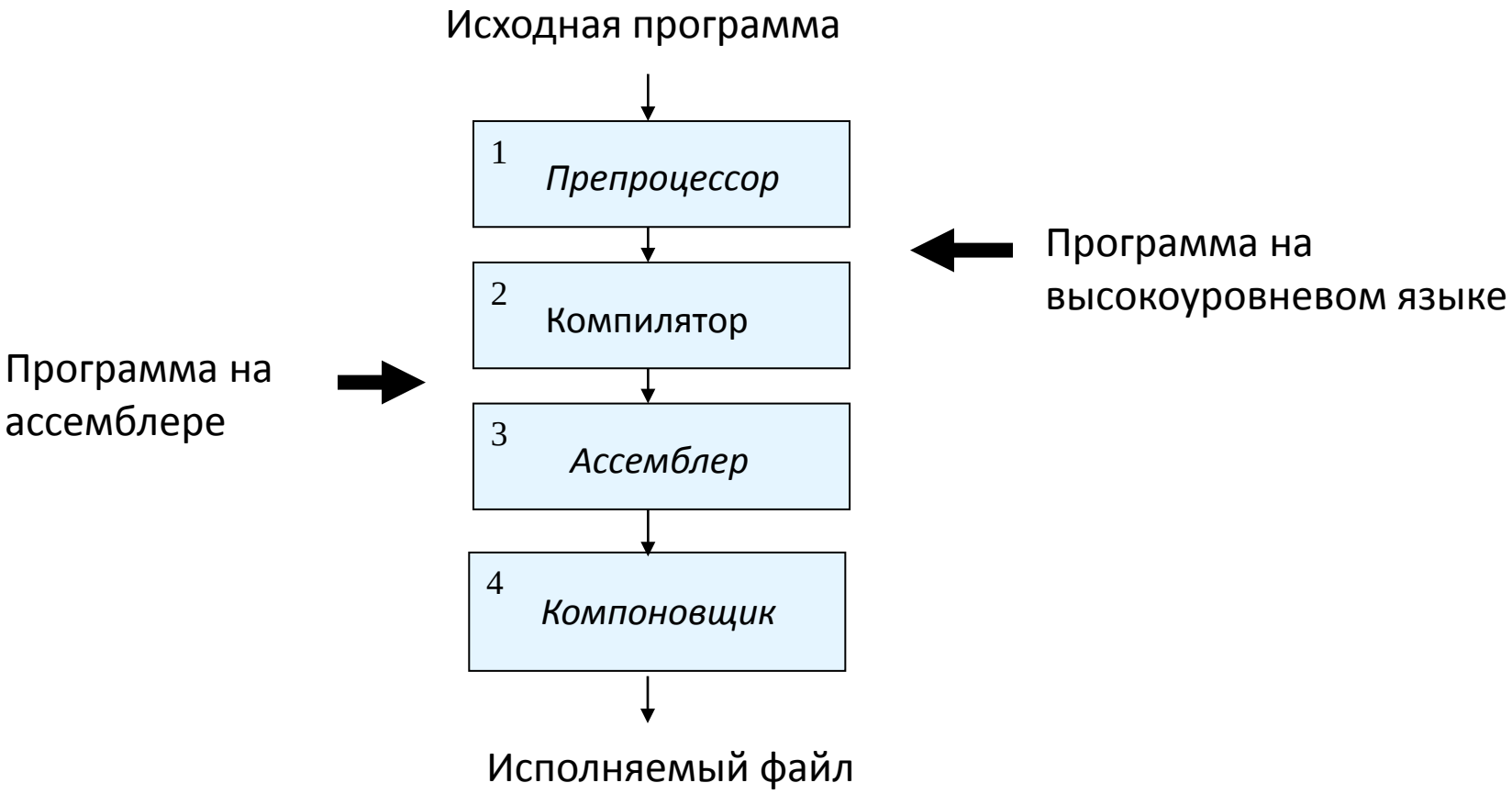


- GCC: C, C++, Fortran, Java->ассемблер целевой машины
- LaTeX: текст с макрокомандами-> .DVI файл
- Компиляторы VHDL: VHDL->схема чипа
- Базы данных: SQL->план выполнения запроса
- PROMT: английский->русский

Оптимизирующие компиляторы

- По программе на исходном языке можно построить множество семантически эквивалентных на целевом языке
- Хотим получить конечную программу, оптимальную по некоторому критерию
  - Скорость выполнения
  - Размер кода (мобильные устройства, I-cache)
  - Энергопотребление (мобильные устройства, кластеры)
  - Размеры “пустых” мест на сверстанной странице (TeX)
- Задача построения оптимальной программы обычно алгоритмически неразрешима

Схема статической компиляции (Си-подобные языки)



Что ещё делает компилятор

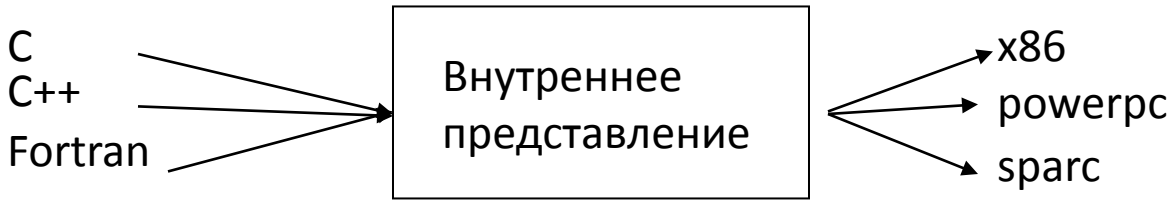
- Диагностика (ошибки и предупреждения)
  - test.c:10: error: 'N' undeclared (first use in this function)
- Выдача отладочной информации
  - gcc –g –o test test.c
  - gdb test
- Выдача “другой” информации (ход оптимизаций, ...)
- Соответствие целевой платформе
  - Память под глобальные и статические данные
  - Подготовка информации для ассемблера и компоновщика
  - ABI (вызовы функций, выравнивание)
  - ...

Модель компиляции

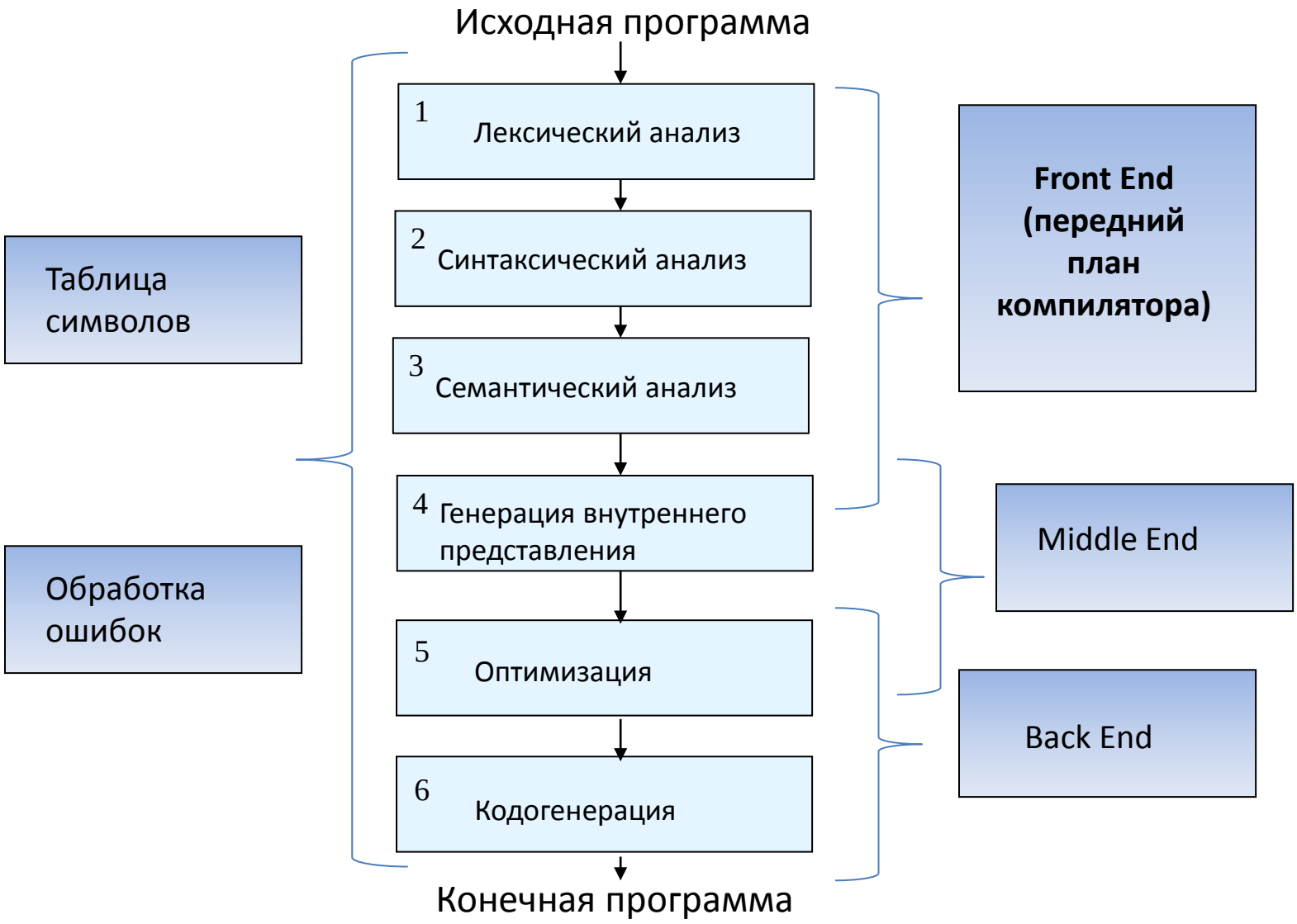
- Два этапа – анализ и синтез
- Анализ: разбор исходной программы, определение выполняемых операций, выдача диагностики, генерация *внутреннего представления* (дерево операций, трехадресный код, …)
- Синтез: генерация эквивалентной программы на целевом языке по внутреннему представлению

Зачем нужно внутреннее представление

- Удобство выполнения анализа и синтеза
- Возможность построить компилятор с нескольких языков на несколько архитектур (gcc)
- Для разных этапов – разные представления



Фазы компиляции



Препроцессирование

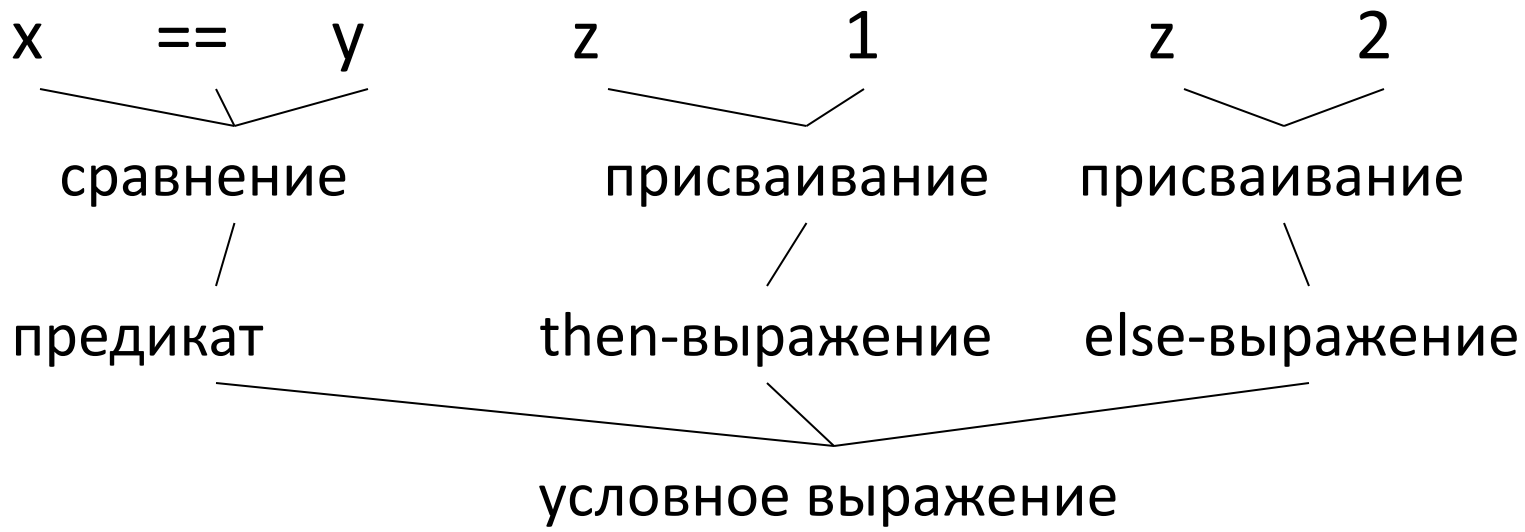
- Сформировать входную программу для компилятора
- Макросы – текстовые подстановки
  - #define i j // Happy debugging!
- Включение файлов
  - #include “headers.h”
- Скрытие деталей реализации
  - # define tolower(c) __tobody (c, tolower, *__ctype_tolower_loc (), (c))

Лексический анализ

- Разбить текст (программу) на слова
- Выполняется за один проход по тексту
- Незначащие символы удаляются
  - Пробелы, комментарии
- Мама мыла раму, а папа чинил машину.
  - “Мама”, “мыла”, “раму”, “,”, “а”, “папа”, “чинил”, “машину”, “.”
- if x == y then z = 1e10; else z = 3.14;
  - if, x, ==, y, then, z, =, 1e10, ;, else, z, =, 3.14, ;

Синтаксический анализ

- Выделить предложения и разобрать их структуру по правилам грамматики языка
- Часто требует рекурсивного обхода дерева, заглядывания вперед на несколько шагов
- if x == y then z = 1; else z = 2;



Семантический анализ

- Предложения языка могут быть многозначными
- Игорь сказал, что Сергей забыл его учебник в общаге
  - Чей учебник забыл Сергей?
- Jack said Jack forgot his textbook at home
  - Сколько всего Джеков?
  - Кто из них забыл учебник?

Семантический анализ

- Грамматика языка дополняется правилами, позволяющими избегать двусмысленных трактовок
- Внутреннее определение переменной Jack перекрывает внешнее
- Необходимы дополнительные проверки этих правил
  - Не больше одного Джека на уровень вложенности
- “Лена забыл дома своё учебник”
  - Несоответствие типов между “Лена” и “забыл”, “учебник” и “своё”

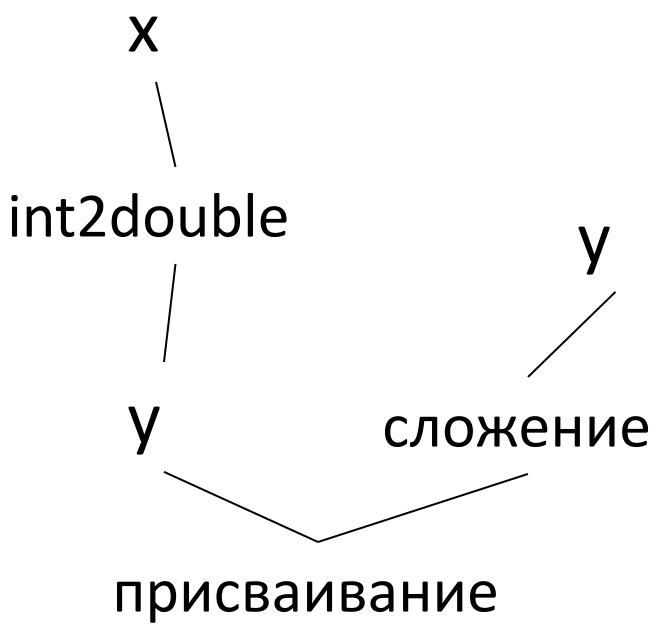
```
{
    int Jack = 3;
    {
        int Jack = 4;
        System.out.
            print(Jack);
    }
}
```

Семантический анализ

- Результирующее дерево может содержать дополнительные операции, вставленные на этом этапе

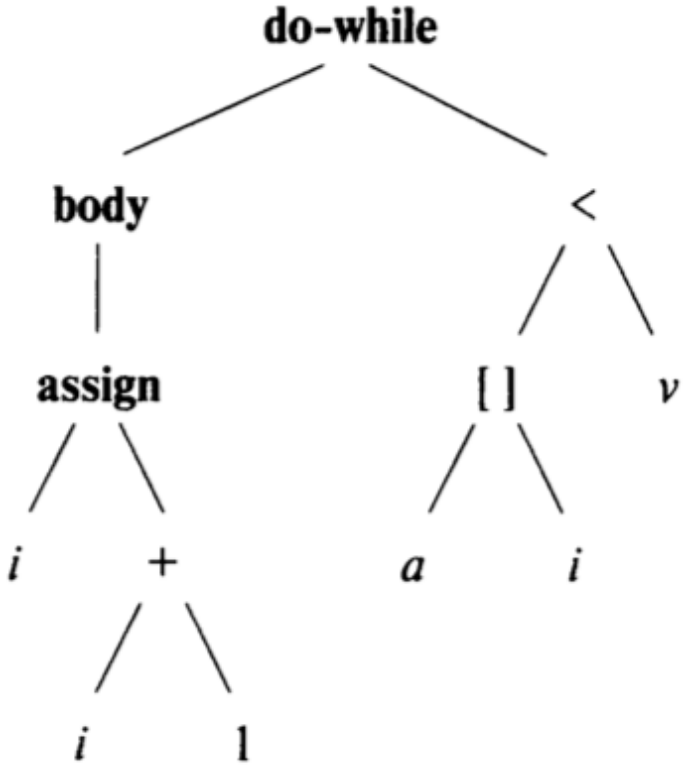
```
int x, double y;

y =  x + y;
```



- Таблица имен заполняется в ходе всего анализа, начиная с лексического

Трансляция во внутреннее представление



Абстрактное Синтаксическое Дерево (AST)

Исходная программа

```
do {
  i = i + 1 ;
} while (a[i] < v);"
```

```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

Результат трансляции: промежуточное представление (трехадресный код)

Кодогенерация

- Получить ассемблер целевой машины по внутреннему представлению
- Выбор инструкций

int x, double y;		mov r1, x; add r1, 20; mov y, r1;
y = x + 20;	или	mov r1, x; mov r2, 20; add r1,r2; mov y,r1;

- Распределение регистров
  - Количество физических регистров машины меньше, чем переменных в программе
  - Для вычислений необходимо использовать промежуточную память
- Поддержка ABI целевой платформы

Оптимизация программы

- Наиболее важная и “наукоемкая” часть (90%)
- Оптимизация обычно состоит из двух частей:
  - Анализ программы – определение необходимых свойств
  - Преобразование программы – поиск выгодных трансформаций и их применение
- Классификация оптимизаций
  - Область применения
    - локальные, глобальные (вся процедура), межпроцедурные
  - Машинно-независимые/машинно-зависимые
    - Выгодны для любой платформы/учитывают особенности целевой платформы (адресная арифметика, регистры, команды)
  - Применение данных о поведении (профиль)
    - Инструментирование (1), запуск и сбор данных, оптимизация (2)
  - Статические (один модуль)/во время компоновки (вся программа)/динамические (JIT)

Оптимизация программы - 2

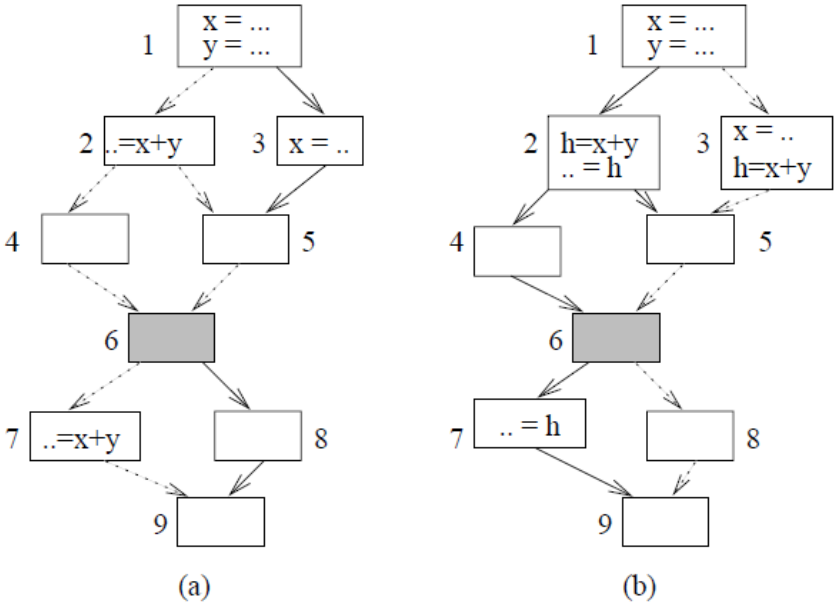
- Машинно-независимые оптимизации
  - Удаление избыточностей (общие подвыражения, нумерация значений)
  - Упрощение графа потока управления
- Цикловые оптимизации
  - Индуктивные переменные (счетчики циклов)
  - Вынос инвариантов
  - Развертка / свертка / слияние / расщепление и т.д.
  - Векторизация (требуются данные о целевой машине)
- Машинно-зависимые оптимизации
  - Распределение регистров
  - Планирование команд и конвейеризация циклов
  - Выбор команд (отображение команд внутреннего представления на команды машины)

Оптимизация программы - 3

- Анализ потоков управления и данных
  - Использование и определение переменных
    – где определяется значение переменной, используемое в данной точке?
  - Анализ указателей (алиасов)
    – что меняет данное присваивание?
  - Анализ циклов и графа потока управления
    (число итераций, вид итерационной функции и т.п.)
- Оптимизации в момент компоновки программы
  - Межпроцедурные оптимизации над несколькими модулями программы
  - Максимальная информация о программе
  - Встраивание функций, распространение данных между функциями, специализация функций

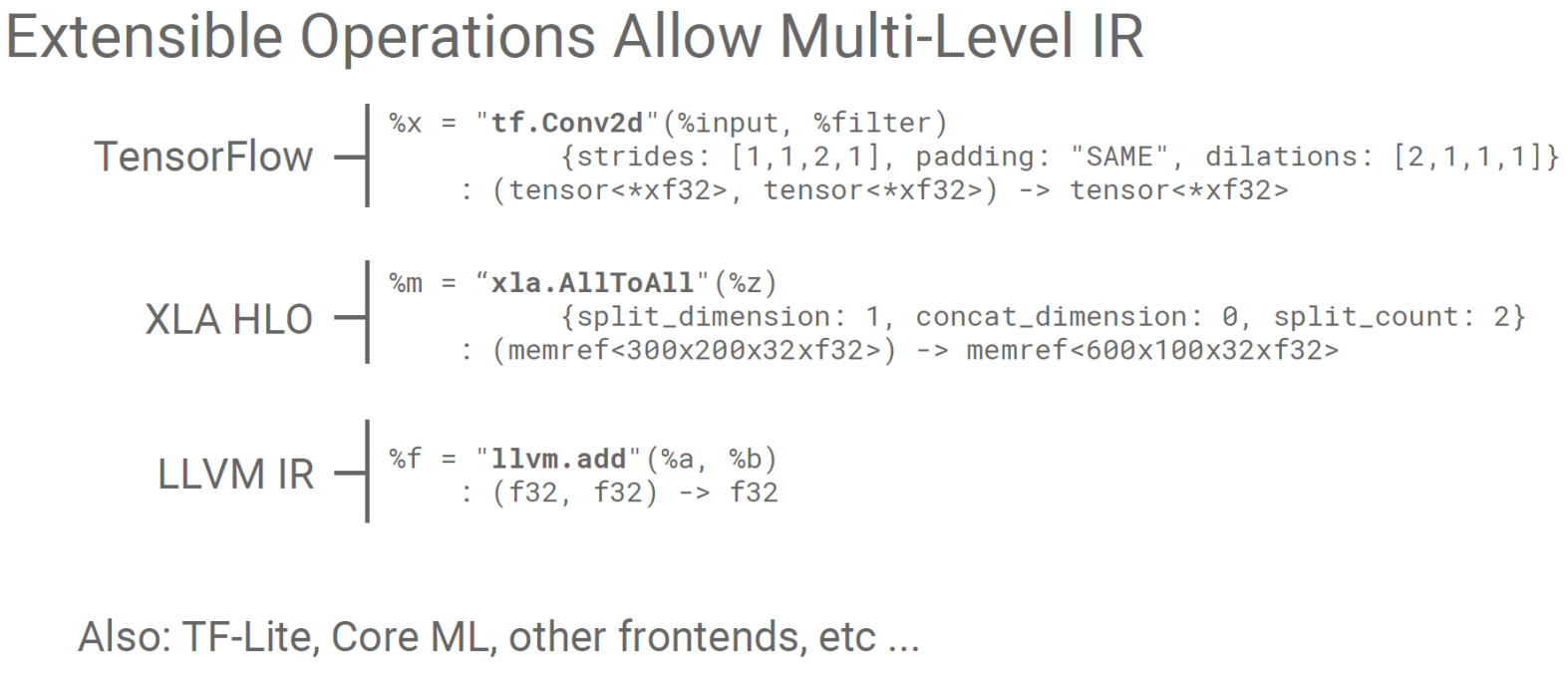
Оптимизации, управляемые профилем

- Обычные оптимизации рассчитаны на “среднее” поведение программы
  - Например, обе ветки условного перехода выполняются с одинаковой вероятностью
- В реальности это верно не всегда
- Решение: собрать информацию о поведении программы на типичных входных данных (“профиль”) и использовать её
- Минус – усложняется процесс компиляции для пользователя



Тренд: многоуровневое представление

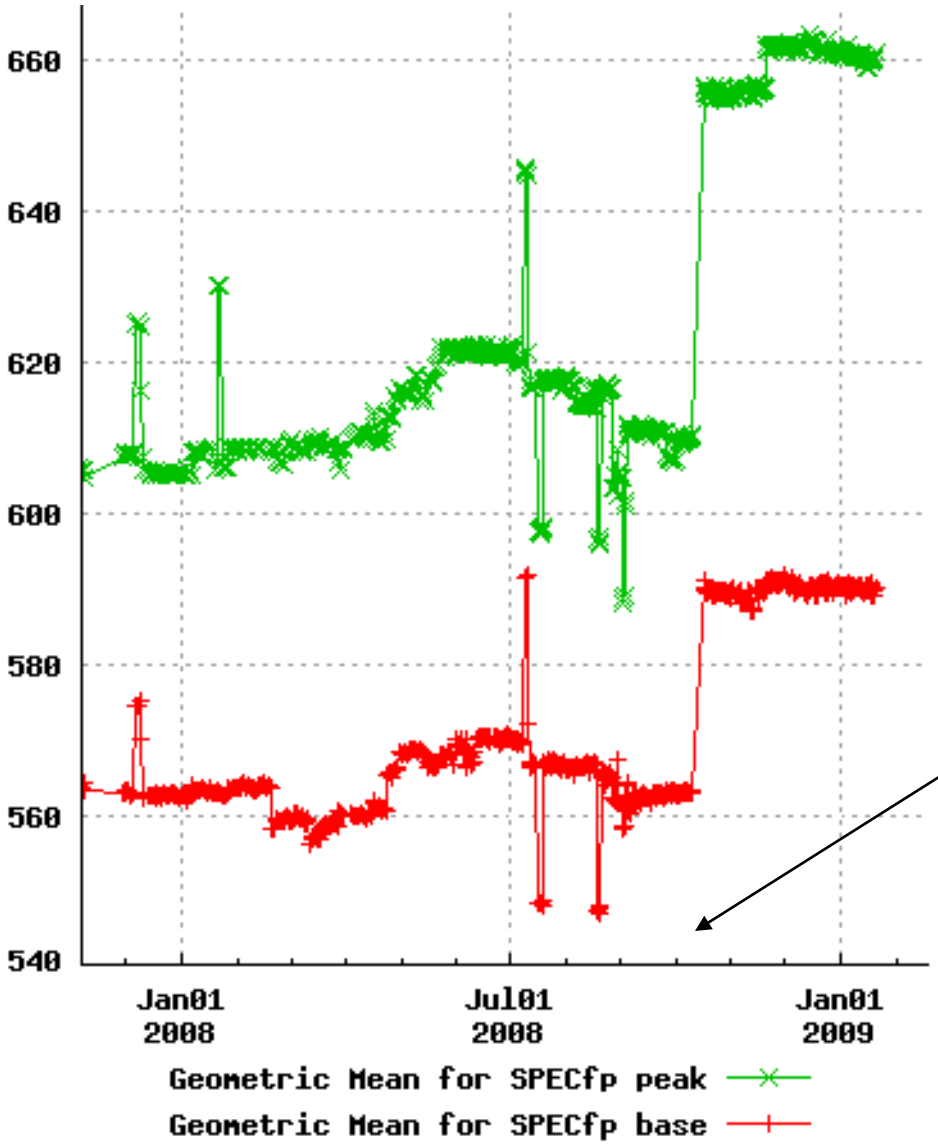
- MLIR позволяет объединить представления разного уровня для языков, составляющих большую систему (TensorFlow)
- Google: https://drive.google.com/file/d/1hUeAJXcAXwz82RXA5VtO5ZoH8cVQhrOK/view
- Конференция C4ML



Пример нашей работы в GCC

- Новый планировщик с поддержкой конвейеризации
  - Поддерживает ряд преобразований команд (спекулятивное и условное выполнение, переименование регистров)
  - Поддерживает конвейеризацию циклов, в том числе вложенных, с неизвестным заранее числом итераций и с большим числом ветвлений
  - Размер патча более 800Кб относительно основной ветви
  - 15 месяцев разработки, 9 месяцев настройки производительности, поддержка кода
  - Включен в GCC 4.4
  - На наборе тестов SPEC FP 2000 получено ускорение в среднем 3.9% (или ~5% с учетом оптимизаций кодогенератора, включенных в GCC ранее), в отдельных тестах до 10%

Результаты планировщика



Включение нескольких новых оптимизаций, в том числе разработанного нами планировщика команд в основную ветвь GCC

Динамические языки и VM

- **Динамические языки**
  - Основные особенности:
    - Управление памятью (сборка мусора, контроль доступа к объектам, границ при обращении к массивам, и т.п.)
    - Динамические типы (классы могут изменяться, а также создаваться новые во время выполнения)
    - Создание нового кода во время выполнения (*eval*)
  - Эти особенности делают статическую компиляцию затруднительной или невыгодной
  - Примеры: JavaScript, Python, Ruby (и в некоторой степени Java)

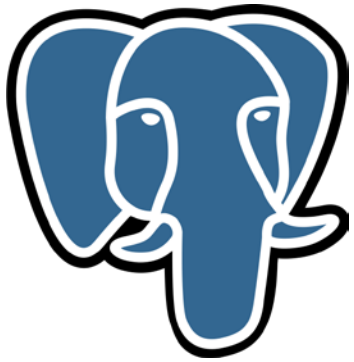
JIT компиляция

- Недостатки:
  - По сравнению с «обычным» (статическим) компилятором, сильно ограничен в сложности выполняемых оптимизаций, т.к. не должен задерживать выполнение программы
- Решение:
  - Оптимизировать только самые «горячие» места
  - Многоуровневый JIT: каждый уровень обрабатывает все более «горячие» места, сложность оптимизаций нарастает
  - Делать сложные оптимизации для следующего уровня параллельно с исполнением неоптимизированного кода на предыдущем уровне

Примеры сделанного

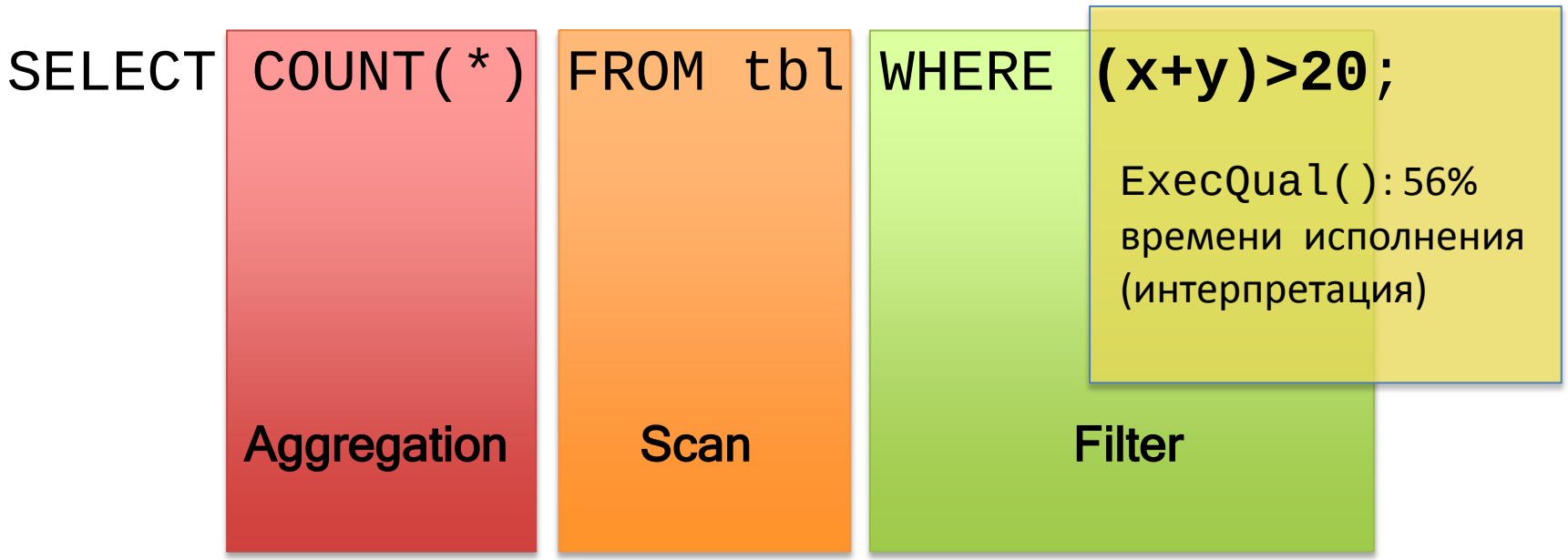
- Оптимизации DFG SFX JIT
  - Определение причин, по которым программа выполняется на неверном уровне JIT
  - Улучшение DFG JIT: добавление поддержки новых операций JIT, новых спекулятивных оптимизаций (~3-5%)
- Предварительная оптимизация JavaScript
  - Двухкомпонентная структура исполнения программ-скриптов:
    - Оффлайн фаза: разбор исходного кода, построение внутреннего представления (bytecode), его оптимизация статическая, либо с использованием профиля. Сохранение в файл.
    - Онлайн фаза: загрузка и выполнение сохраненного внутреннего представления
- Оптимизация эмулятора QEMU (двоичная трансляция)
  - Это тоже JIT-компиляция, но на внутреннем представлении, полученном из бинарного кода
  - Оптимизации из компиляторов «в лоб» не работают

Ускорение PostgreSQL



- Что именно мы хотим ускорить?
  - Сложные запросы, где «узким местом» в производительности является процессор, а не дисковые операции
    - OLAP, поддержка принятия решений, и т.д.
  - Цель: оптимизация производительности на наборе тестов TPC-H
- Как ускорить?
  - Использовать LLVM JIT, на первом этапе – для компиляции выражений в операторе WHERE

Пример оптимизации запроса



Вычисление выражений

(x+y)>20

Генератор LLVM-биткода

Биткод LLVM

```
define i32 @where_expr(i32 %x, i32 %y) #0 {
entry:
  %add = add nsw i32 %y, %x
  %cmp = icmp sgt i32 %add, 20
  %conv = zext i1 %cmp to i32
  ret i32 %conv
}
```

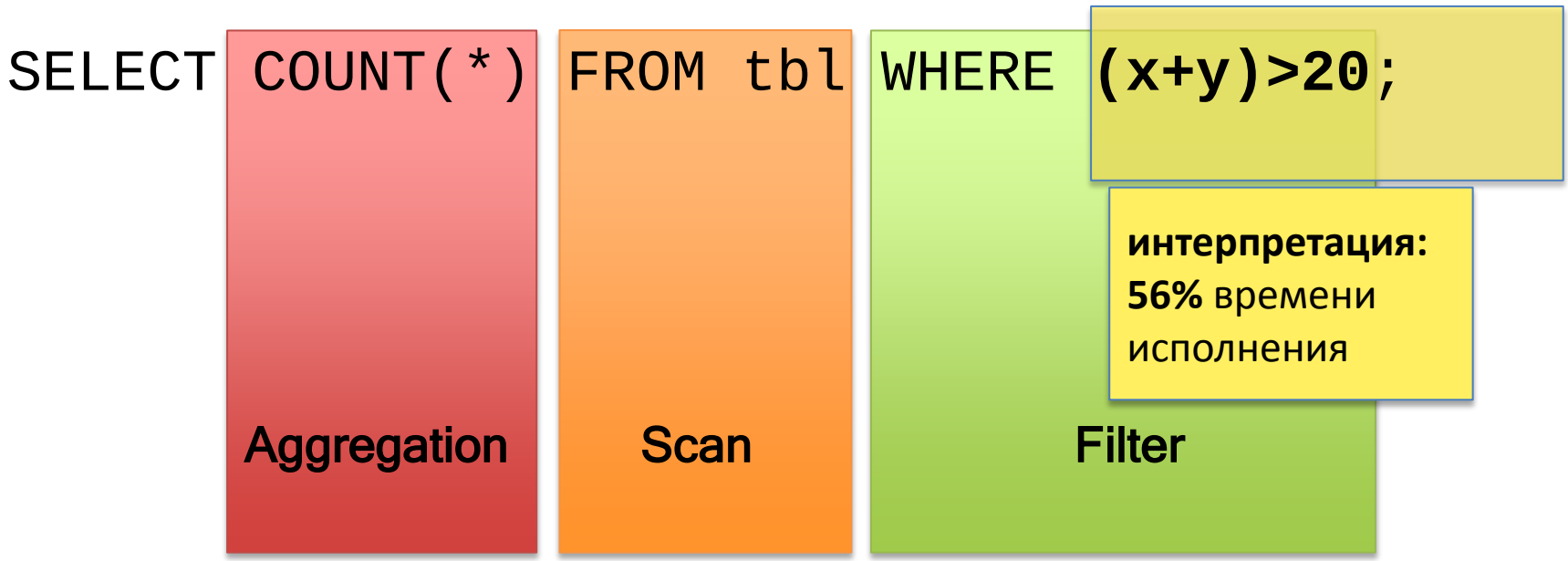
LLVM MCJIT

Интерпретация vs **Оптимизированный двоичный код**

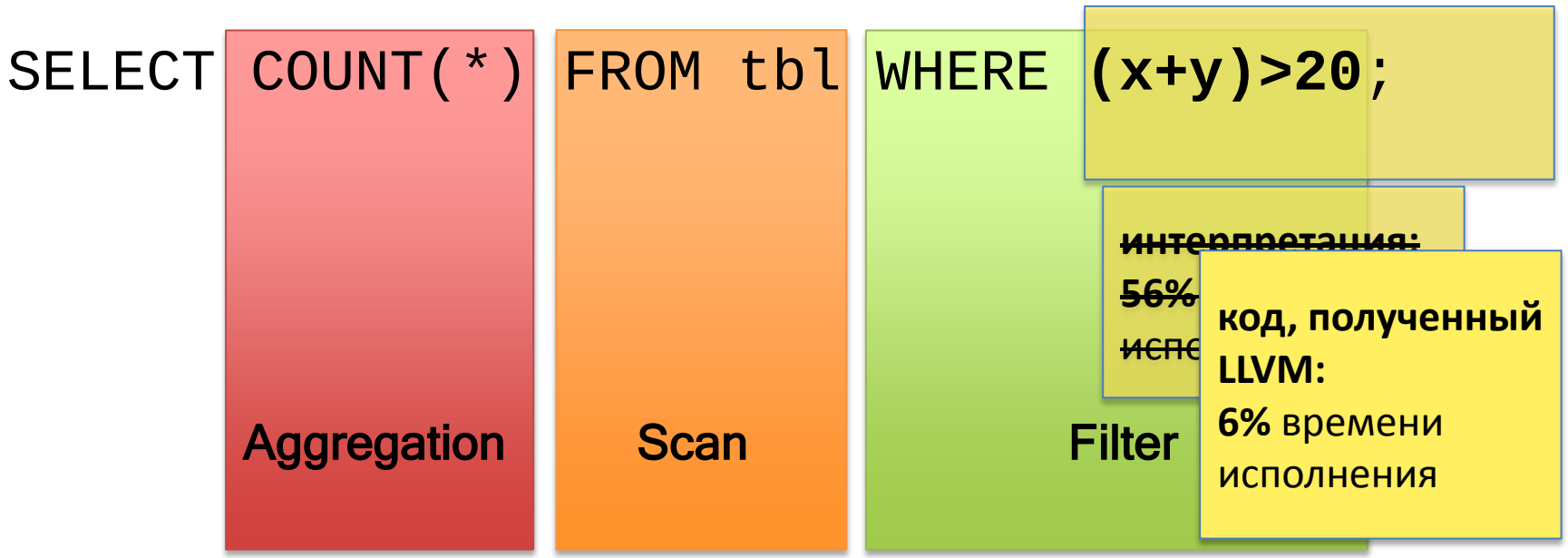
```
foo:
        addl %esi, %edi
        cmpl $20, %edi
        setg %al
        movzbl    %al, %eax
        retq
```

в ~10 раз быстрее

Пример оптимизации запроса



Пример оптимизации запроса



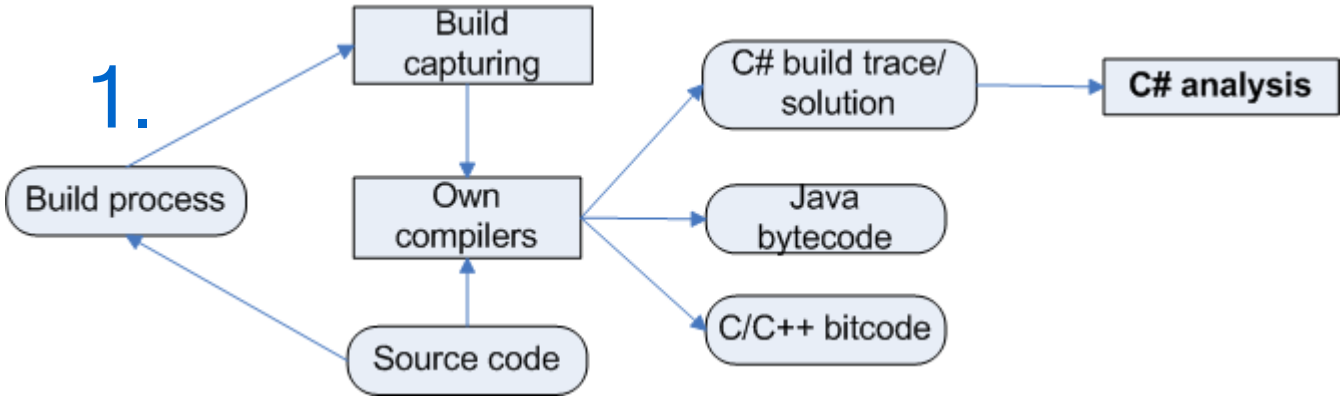
=> Ускорение выполнения запроса в 2 раза

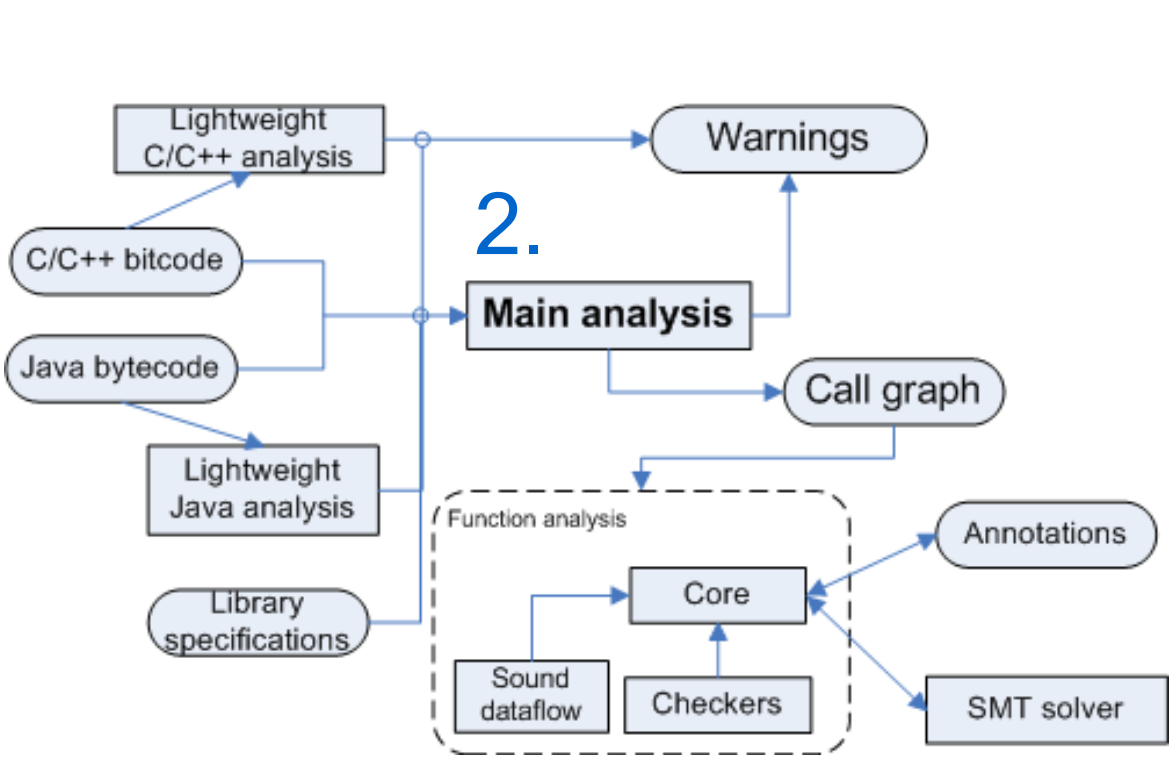
Static Analysis Requirements

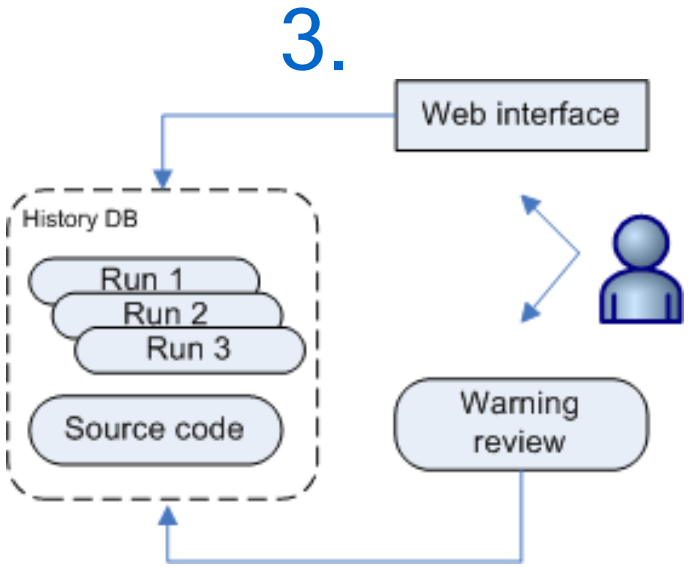
ISP RAS

- Wide applicability: defect detection, program understanding, performance, ...
- Application for secure development lifecycle
  - On development phase (nightly builds) or on Q&A phase
- Requirements that follow:
  - Fully automatic analysis (no need to change the code)
  - Scalable to millions of LOC
  - Fair percent of true positives (>60%)
  - Support of programming languages (C/C++/Java/...), defect types (many), environments (Windows/Linux)
  - Extensibility with new checkers, flexibility (tailored config)
  - CI integration

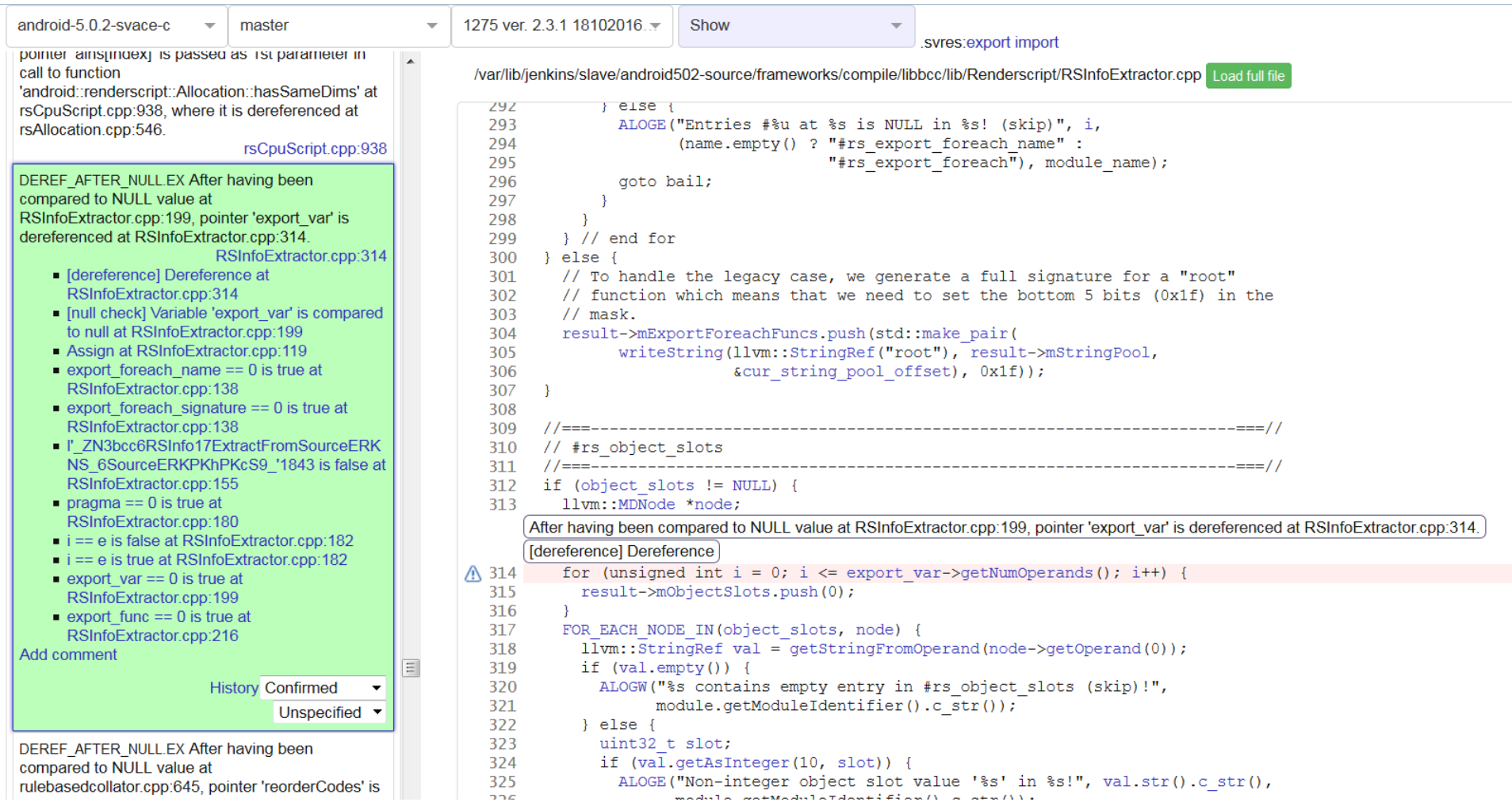
Svace Architecture







Svace GUI Example



Выводы

- Много актуальных задач, где применяются компиляторные технологии
  - Оптимизация программ, создание компиляторов для новых архитектур, анализ на выявление ошибок или других свойств, подсказки разработчику, …
- Учиться лучше всего на реальных проектах
  - «Что такое state of the art и как им пользоваться» передается только показом, а не рассказом
- В этой области ИСП РАН – один из лучших в России
  - Отдел компиляторных технологий – более 100 человек, выполняемые работы очень разнообразны
- Вопросы? → abel@ispras.ru