

# **ИНФОРМАЦИОННО-УПРАВЛЯЮЩИЕ СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ**

## **Лекция 6:**

### ***Оценка наихудшего времени выполнения программ (WCET) - 2***

Кафедра АСВК,  
Лаборатория Вычислительных Комплексов  
Балашов В.В.

# Почему важен WCET

- *Время выполнения* программ играет ключевую роль при анализе систем реального времени
- Это время, например, встречается в формуле:

Наихудшее время отклика

Период

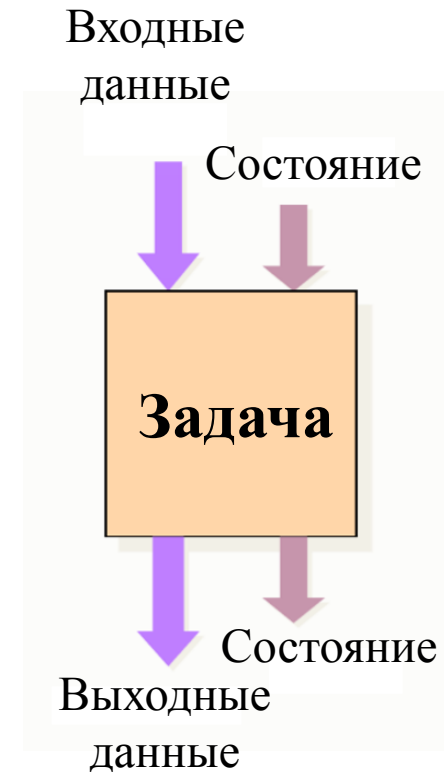
Наихудшее время выполнения

$$R_i = C_i + \sum_{j \in hp(i)} \lceil R_i / T_j \rceil C_j$$

Откуда берутся значения  $C_i, C_j$ ?

# Простейшая вычислительная задача

- Входные данные доступны в момент старта
- Выходные данные готовы в момент завершения
- Нет блокировок в процессе выполнения
- Нет синхронизации или обмена данными в процессе выполнения
- Время выполнения зависит только от:
  - входных данных
  - состояния задачи в момент старта (внешние воздействия отсутствуют)



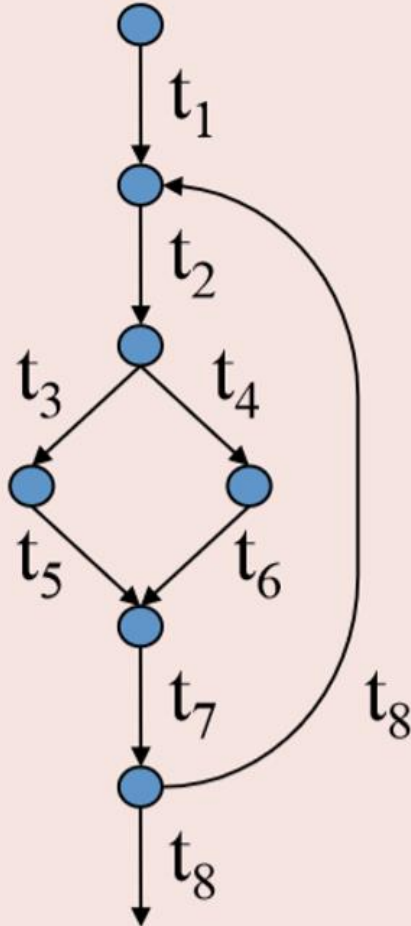
# Наихудшее (максимальное) время выполнения

Наихудшее время выполнения программного кода (worst case execution time, WCET) – это максимальное время, которое требуется для выполнения

- данного фрагмента кода
- в данном контексте (входные данные, состояние)
- на заданном аппаратном вычислителе

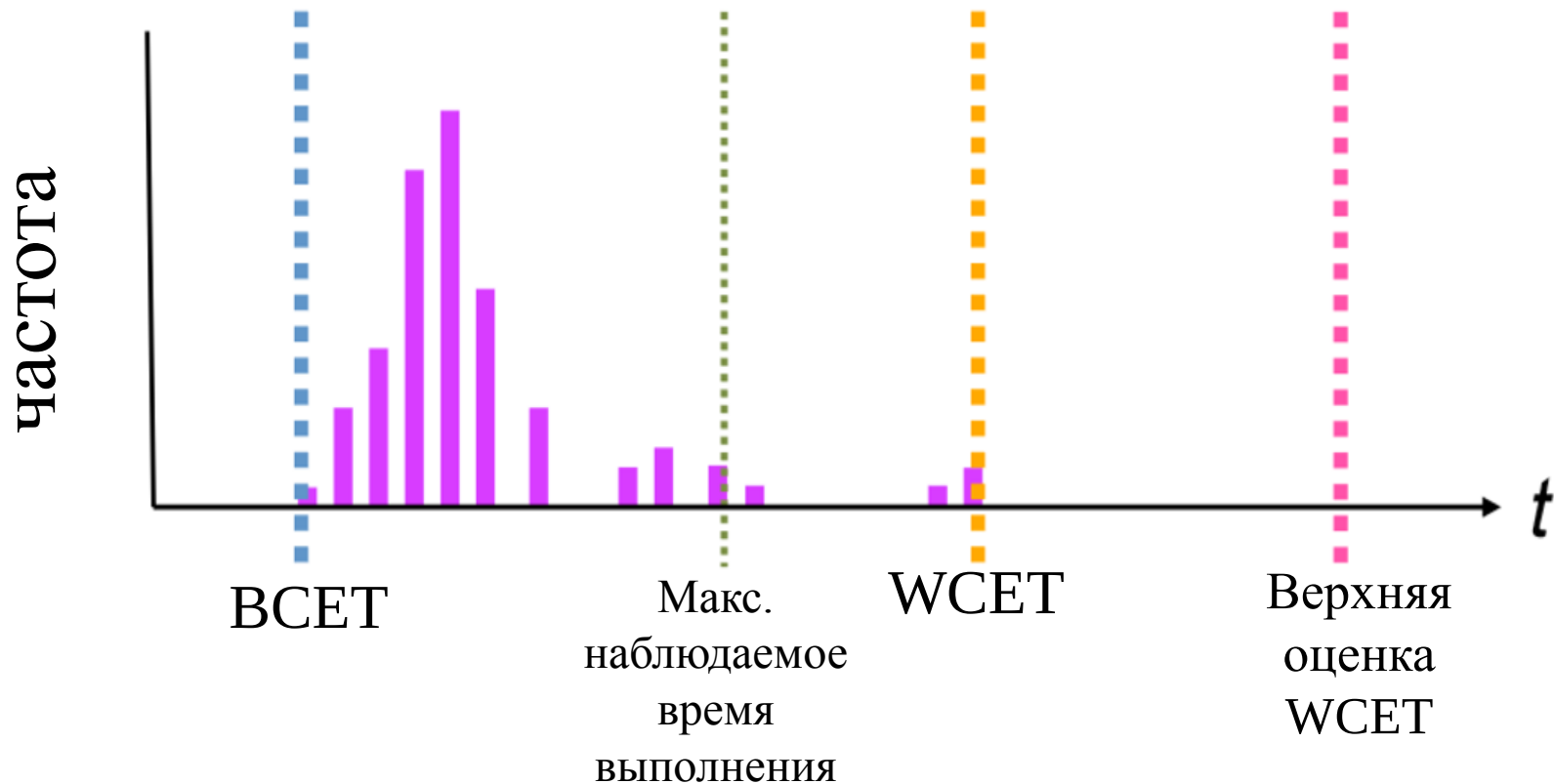
# Чем определяется WCET

## Задача



- Возможные **последовательности действий** задачи (пути выполнения)
- **Длительность** выполнения каждого действия на каждом допустимом (т.е. практически возможном) пути выполнения

# Распределение времён выполнения



Множество различных времён выполнения

Нетривиальный анализ путей выполнения

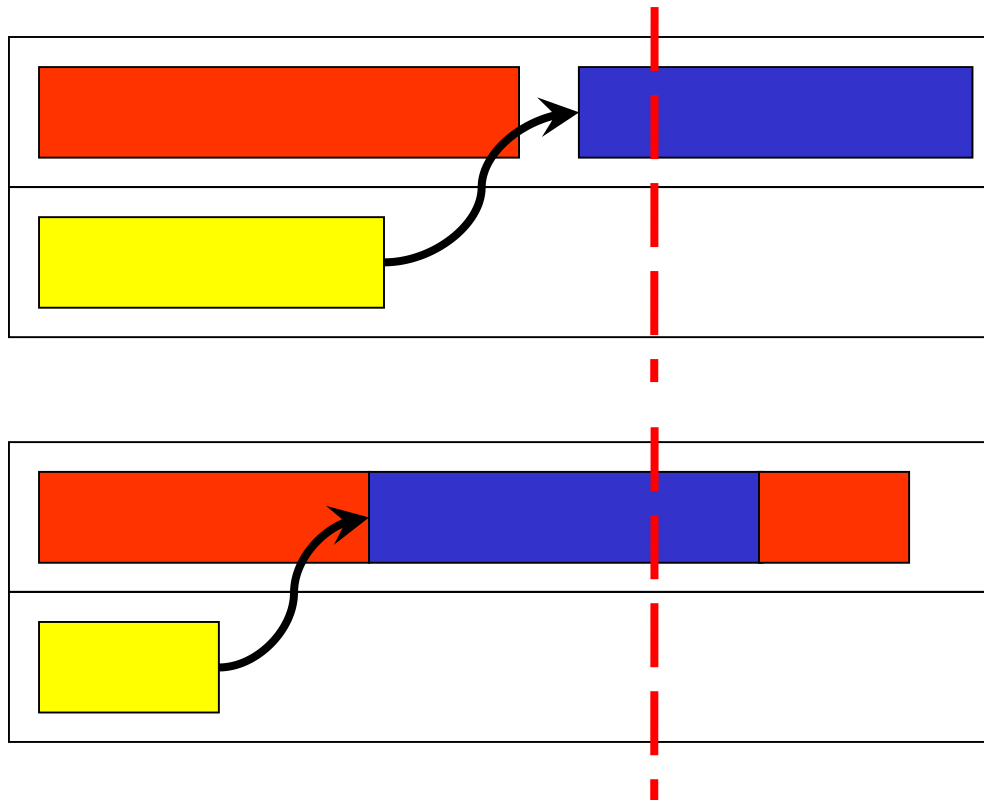
Сложное моделирование задержек от аппаратуры

# Оптимизация программного кода для временной предсказуемости

# Что важнее всего в таймингах работы RV-систем?

- В первую очередь – предсказуемость
  - Предсказуемый WCET
  - Также – предсказуемый BCET
- Производительность – во вторую очередь
- => потребность во временной предсказуемости:
  - Аппаратура
  - Программное обеспечение

# Почему важен VCET



Желтая задача:  $VCET \ll WCET$

➔ красная задача нарушает директивный срок

# Performance Criteria

Write program  $\pi$  with best performance

MEAN

traditional: minimize  $\sum_i p_i \cdot c(\pi, \delta_i, \mu)$

probability of input  $\delta_i$

machine

input data

WCET

HRT: minimize  $\max_i c(\pi, \delta_i, \mu)$

# Традиционное программирование

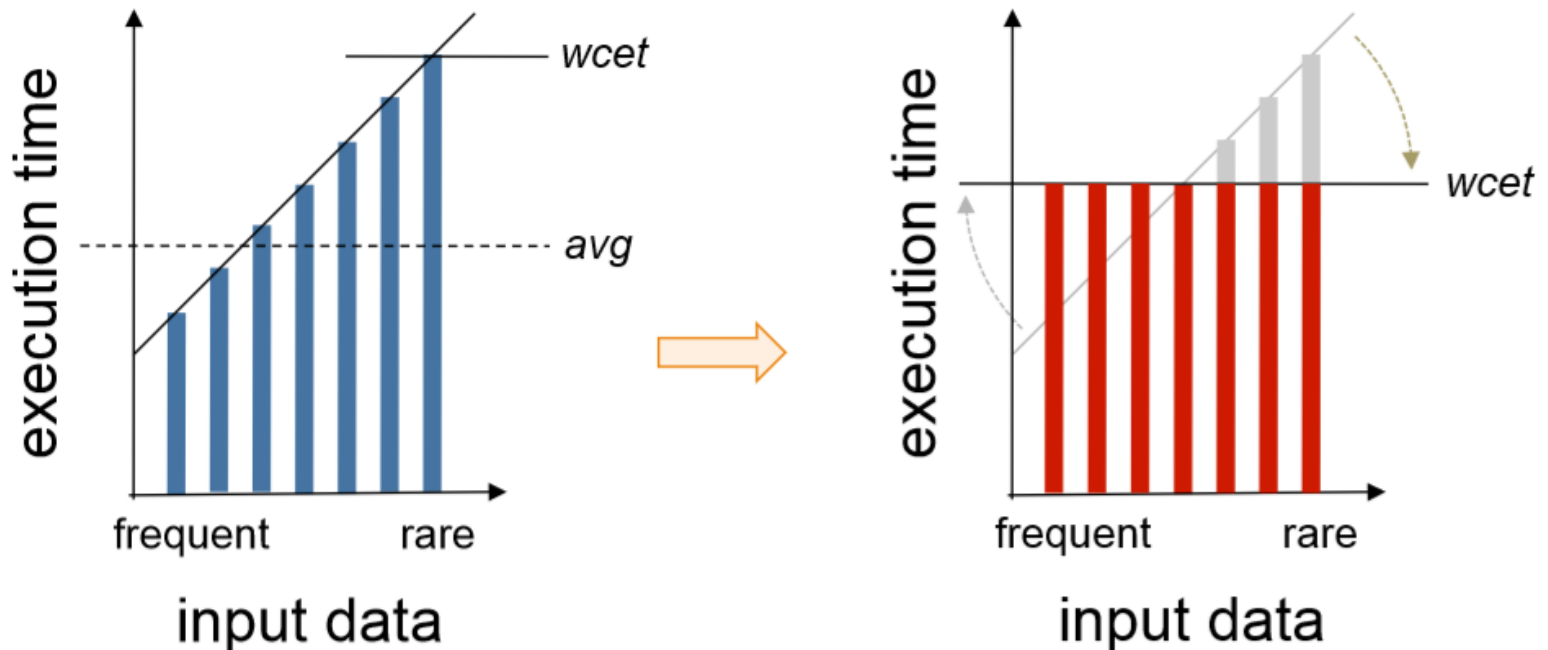
- Цель: хорошая производительность в среднем
- Стратегия: профилировать и оптимизировать производительность на наиболее часто встречающихся наборах входных данных => возможное ухудшение WCET

Причина: сложность неравномерно распределена между наборами входных данных; оптимизация частых сценариев приводит к ухудшению производительности на редких сценариях

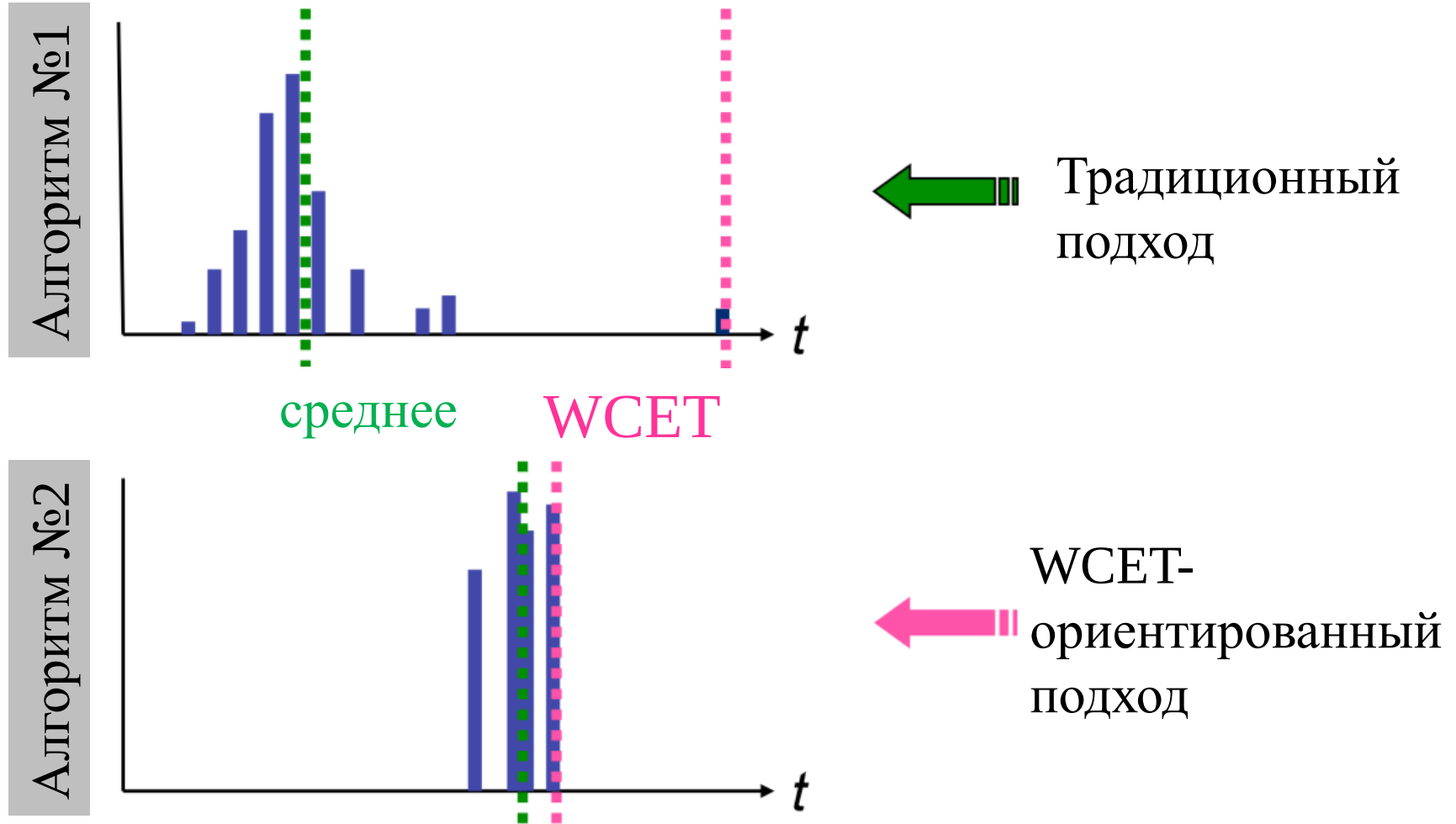
# WCET-ориентированное программирование

- ❑ Стремимся освободить код от ветвлений в зависимости от входных данных
- ❑ Минимизируем число действий, выполняемых только для отдельных наборов входных данных

# Традиционное vs. WCET-ориентированное программирование



# Критерий производительности



# Эксперименты

- Традиционный vs. WCET-ориентированный код
- Ветвящийся vs. линейный код

Реализованные алгоритмы:

- Пузырьковая сортировка
- Поиск первого вхождения
- Двоичный поиск
  - одно/два сравнения на каждой итерации

```

static int binSearch_avg(int key, int a[])
{
    int left = 0, right = SIZE-1, idx, inc;
    int found = 0;

    do
    {
        idx = (right + left) >> 1;
        if (a[idx] == key)
        {
            found = 1;
        }
        else if (a[idx] < key)
        {
            left = idx+1;
        }
        else
        {
            right = idx-1;
        }
    } while (!found && (right >= left));

    if (found)
    {
        return idx;
    }
    else
    {
        return -1;
    }
}

```

```

static int binSearch_wcet(int key, int a[])
{
    int left = 0, right = SIZE-1, idx, inc;

    idx = (right + left) >> 1;

    for(inc = SIZE; inc > 0; inc = inc >> 1)
    {
        right = (key < a[idx] ? idx - 1 : right);
        left = (key > a[idx] ? idx + 1 : left);
        idx = (right + left) >> 1;
    }

    return idx;
}

```

# Эксперименты

- Традиционное vs. WCET-ориентированное программирование

| Алгоритм    | Традиционное |      | WCET-ориентированное |      |
|-------------|--------------|------|----------------------|------|
|             | Среднее      | WCET | Среднее              | WCET |
| Bubble*     | 599          | 724  | 609                  | 663  |
| Find-first* | 68           | 122  | 103                  | 103  |
| Bin-search  | 94           | 124  | 105                  | 106  |

\* В традиционном варианте: без goto; используется признак раннего завершения

# Свойства WCET-ориентированных программ

- **WCET меньше**, чем у традиционных программ
- **Анализ путей проще** в связи с устранением/сокращением зависимостей от входных данных
- **Оценка WCET проще** из-за меньшего числа путей выполнения (или единственного пути)
- **Меньше разброс** времен выполнения => более стабильна работа системы в целом
- **Низкая сложность** потока управления => хорошо подходит для управляющих программ, критичных для безопасности
- Нетрадиционные алгоритмы

# Стремимся к временной предсказуемости

Сокращаем зависимость порядка действий от  
внешних факторов

⇒ Линейный (single-path) код

- Нет ветвлений по входным данным
- Предикатное выполнение
- Акцент на потоке данных (а не на потоке управления)

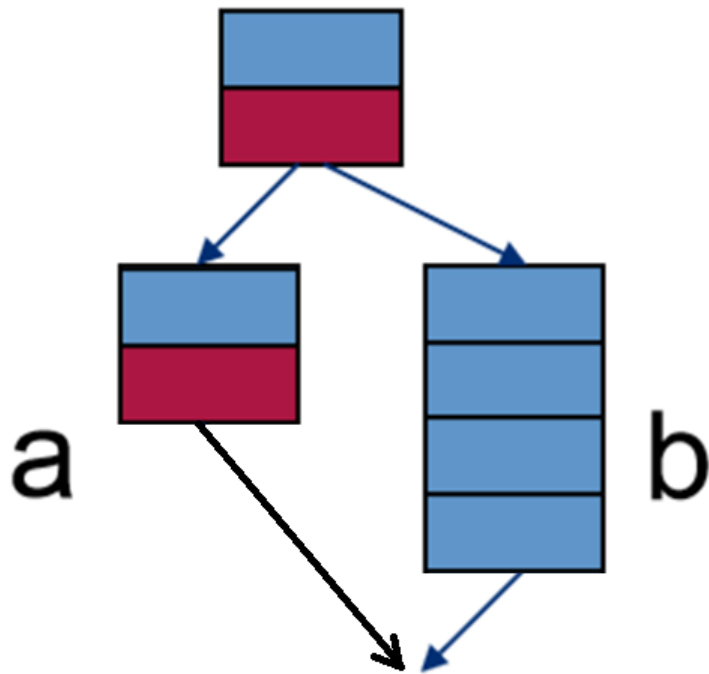
# WCET-ориентированное программирование + линеаризация кода

- Константное время выполнения
- Низкий WCET
- Тривиальный анализ путей
- Простой анализ WCET

⇒ Полная временная предсказуемость

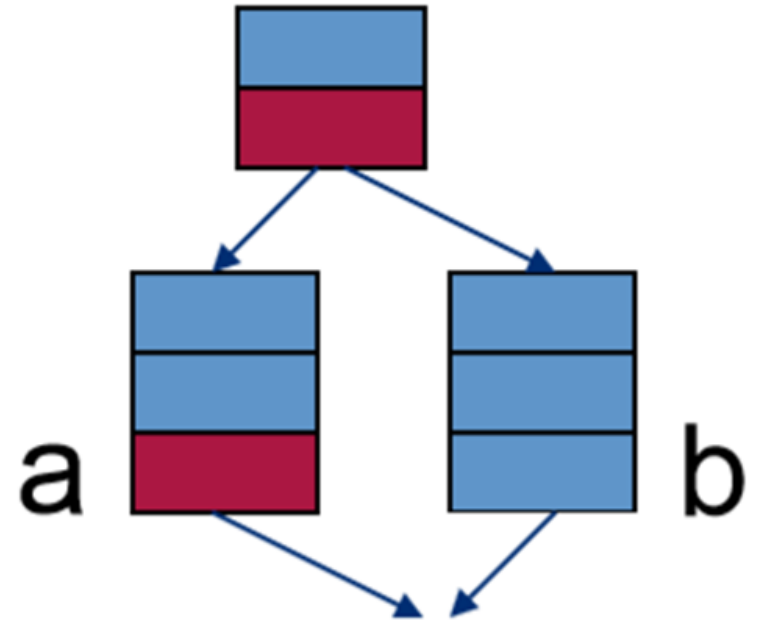
⇒ Приемлемая производительность

Линеаризация: код, оптимизированный под  
средний случай, замедляется сильнее



$$T_{avg}: 4,1 \rightarrow 8$$

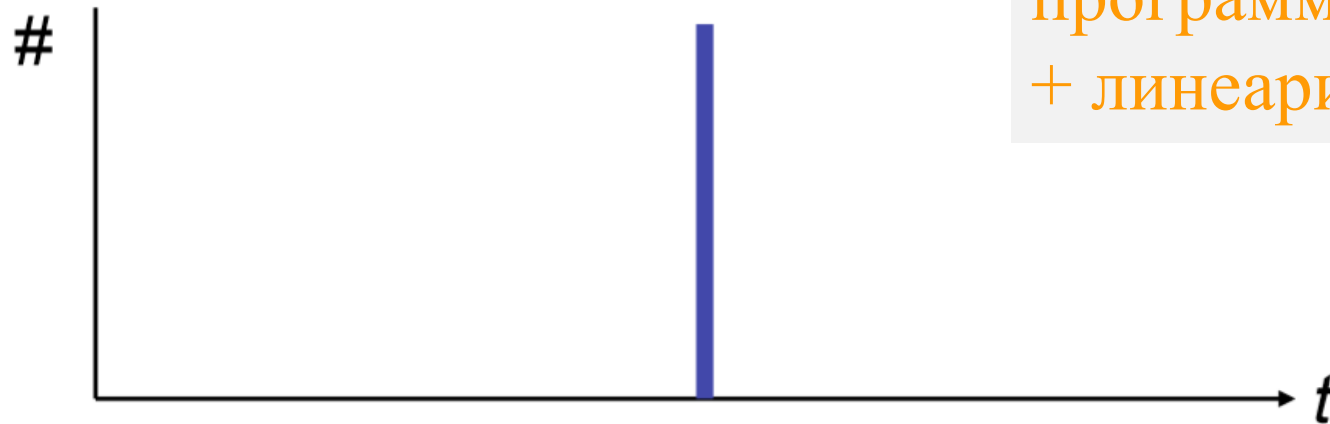
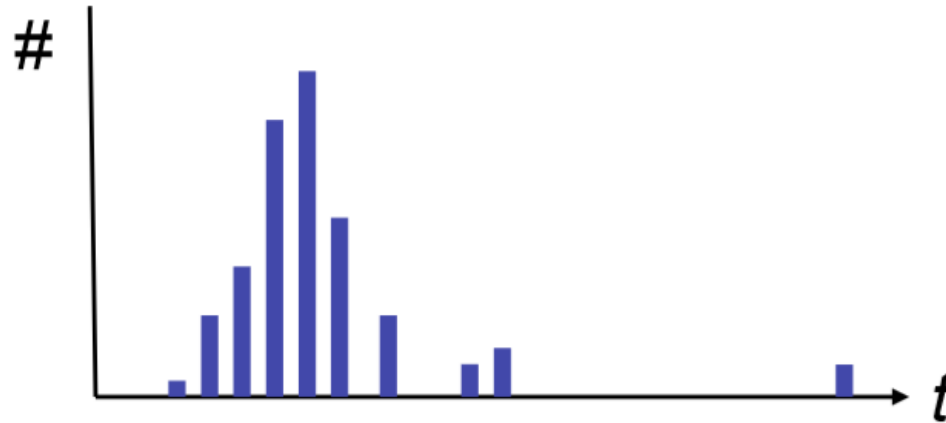
(x1,95)



$$T_{avg}: 5 \rightarrow 8$$

(x1,6)

# Времена выполнения

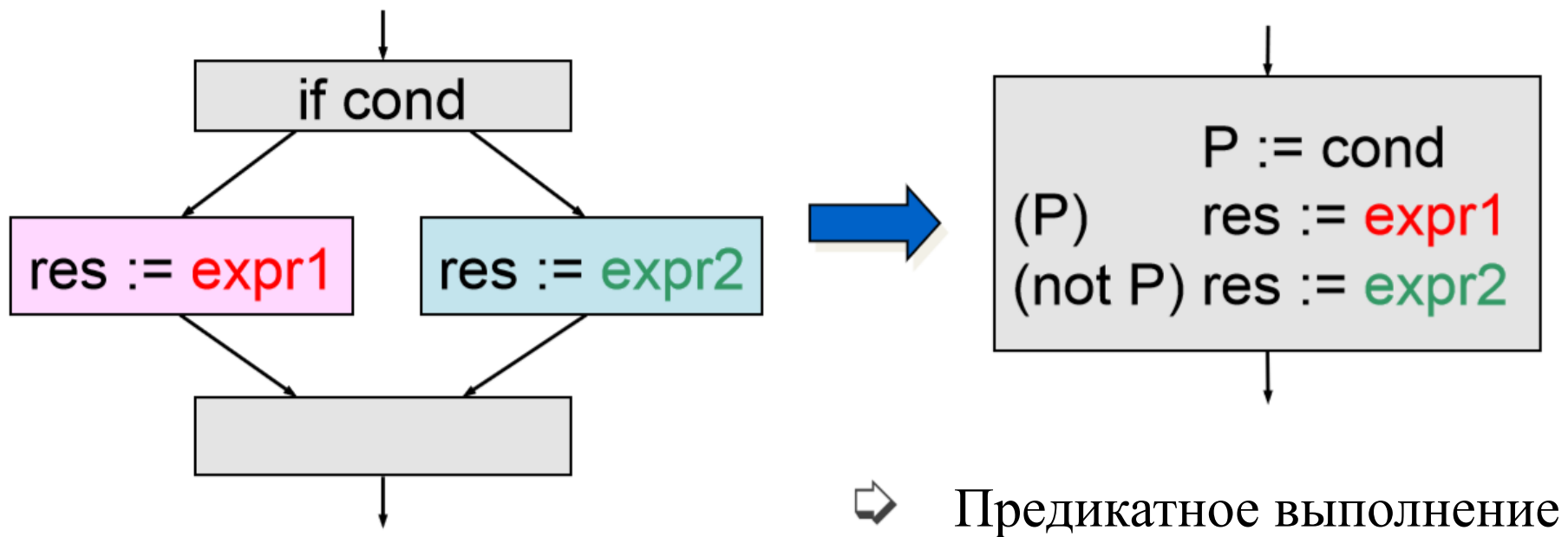


Времена  
выполнения кода до  
и после перехода на  
WCET-  
ориентированное  
программирование  
+ линеаризации

Как получить линейный код?

# Устранить зависимости по данным из потока управления

- Аппаратура с константными задержками
- Линеаризованный код



# Предикатное выполнение

*...это условное выполнение или невыполнение команды в зависимости от значения булевой переменной, называемой предикатом  
[Hsu et al. 1986]*

## Выполнение предикатной команды:

- Безусловная выборка команды из памяти
- Предикат истинен  $\Rightarrow$  нормальное выполнение команды
- Предикат ложен  $\Rightarrow$  команда не изменяет состояние процессора

# Ветвящийся vs. предикатный код

Пример кода:

**if** rA < rB **then** swap(rA, rB);



Ветвящийся код

```
cmplt rA, rB
bf     skip
swp    rA, rB
skip:
```



Предикатный код

```
predlt Pi, rA, rB
(Pi) swp    rA, rB
```

# Аппаратная поддержка предикатного выполнения

Предикатные регистры

Команды для работы с предикатами (определить, установить, сбросить, загрузить, сохранить)

- Предикатные команды
  - Поддержка **полной предикатности**: выполнение любых команд управляется предикатами
  - Поддержка **частичной предикатности**: лишь некоторое подмножество команд могут быть предикатными (например, копирование данных, присваивание значений)

# Специфика частичной предикатности

## Опережающее выполнение вычислений

- Команды, не являющиеся предикатными, выполняются безусловно
- Их результаты сохраняются во временных переменных
- В дальнейшем предикат определяют, содержание какой из временных переменных следует использовать

Пример:

|            |                 |
|------------|-----------------|
| (pred)     | cmov dest, src1 |
| (not pred) | cmov dest, src2 |

**Внимание**: команды, вычисляющие значения временных переменных, не должны генерировать исключения (пример: деление на ноль, обращение к недопустимому адресу памяти).

# Полностью vs. частично предикатный код

Оригинальный код

```
if src2 ≠ 0 then dest := src1 / src2;
```



Полностью  
предикатный код:

```
Pred := (src2 ≠ 0)
```

```
(Pred) div dest, src1, src2
```

# Fully vs. Partially Predicated Code (2)

Original code:

```
if src2  $\neq$  0 then dest := src1 / src2;
```



Partially predicated code, first attempt:

may raise  
an exception  
on division  
by zero

```
Pred := (src2  $\neq$  0)
```

```
(Pred)
```

```
div    tmp_dst, src1, src2  
cmov   dest, tmp_dst
```

# Fully vs. Partially Predicated Code (3)

Original code:

```
if src2  $\neq$  0 then dest := src1 / src2;
```



Partially predicated code:

if src2 equals 0,  
then replace it  
by a safe value  
(e.g., 1) to avoid  
division by zero

|            |                                                  |
|------------|--------------------------------------------------|
|            | Pred := (src2 $\neq$ 0)                          |
| (not Pred) | mov tmp_src, src2<br>cmov tmp_src, \$safe_val    |
| (Pred)     | div tmp_dst, src1, tmp_src<br>cmov dest, tmp_dst |

# Минимальная поддержка предикатности

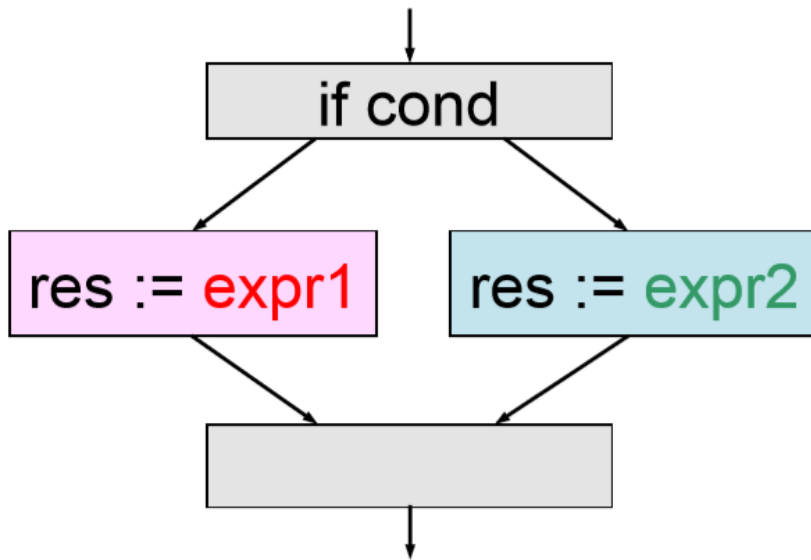
- Команда условного копирования:

```
movCC destination, source
```

Семантика:

```
if CC  
then destination := source  
else no operation
```

# If-преобразование с условным копированием



```
t1 := expr1'  
t2 := expr2'  
test cond  
movT res, t1  
movF res, t2
```

} Избегать  
побочных  
эффектов

# Эмуляция условного копирования

- В архитектурах без поддержки предикатности, условное копирование можно эмулировать при помощи операций с битовыми масками.

Пример: **if (cond) x=y; else x=z;**



```
t0 = 0 – cond;      // fat bool: 0..false, -1..true
t1 = ~t0;           // bitwise negation (fat bool)
t2 = t0 & y;
t3 = t1 & z;
x = t2 | t3;
```

Допущение: переменные всех  
используемых типов имеют  
одинаковый размер

# Пример:

```
for(i=SIZE-1; i>0; i--)  
{  
  for(j=1; j<=i; j++)  
  {  
    if (a[j-1] > a[j])  
    {  
      t = a[j];  
      a[j] = a[j-1];  
      a[j-1] = t;  
    }  
  }  
}
```



Пузырьковая сортировка: на входе массив  $a[SIZE]$

```
for(i=SIZE-1; i>0; i--)  
{  
  for(j=1; j<=i; j++)  
  {  
    t1 = a[j-1];  
    t2 = a[j];  
  
    (t1>t2): t = a[j];  
    (t1>t2): a[j] = a[j-1];  
    (t1>t2): a[j-1] = t;  
  }  
}
```

# Пример:

Пузырьковая сортировка: на входе массив  $a[\text{SIZE}]$

```
for(i=SIZE-1; i>0; i--)  
{  
  for(j=1; j<=i; j++)  
  {  
    if (a[j-1] > a[j])  
    {  
      t = a[j];  
      a[j] = a[j-1];  
      a[j-1] = t;  
    }  
  }  
}
```



```
for(i=SIZE-1; i>0; i--)  
{  
  for(j=1; j<=i; j++)  
  {  
    cond = (a[j-1] > a[j]);  
  
    (cond): t = a[j];  
    (cond): a[j] = a[j-1];  
    (cond): a[j-1] = t;  
  }  
}
```

# Пример:

Пузырьковая сортировка: на входе массив  $a[SIZE]$

```
for(i=SIZE-1; i>0; i--)  
{  
  for(j=1; j<=i; j++)  
  {  
    if (a[j-1] > a[j])  
    {  
      t = a[j];  
      a[j] = a[j-1];  
      a[j-1] = t;  
    }  
  }  
}
```



```
for(i=SIZE-1; i>0; i--)  
{  
  for(j=1; j<=i; j++)  
  {  
    t1 = a[j-1];  
    t2 = a[j];  
  
    test (t1>t2);  
    movt: a[j-1] = t2;  
    movt: a[j] = t1;  
  }  
}
```

# Свойства линейного кода

- Каждое выполнение порождает одну и ту же трассу команд, т.е. одну и ту же последовательность обращений к памяти команд.
- Анализ путей тривиален – существует единственный путь.
- Два выполнения, начинающиеся с одинакового состояния кэша команд, порождают одинаковые последовательности промахов/попаданий в кэш команд.

# Линейный код и задержки

Каждое выполнение порождает одну и ту же последовательность (а также количество) выполненных инструкций -> хорошая основа для получения фиксированного времени выполнения.

Команды с неконстантным, зависящим от данных, временем выполнения вызывают флуктуации времени выполнения кода.

Разные начальные состояния памяти (в т.ч. кэша) могут привести к различным задержкам при доступе к памяти команд и данных, а значит к различным временам выполнения кода.

# Обеспечение константного времени выполнения

Не позволяем внешним факторам влиять на:

- Последовательность выполняемых команд
  - Длительность команд
- 
- Всегда начинать с одного и того же состояния кэша команд, конвейера, логики предсказания ветвлений и т.п.
  - Обеспечивать константные задержки при доступе к данным
  - Обеспечивать константные длительности всех операций процессора
  - Все внешние воздействия (в т.ч. вытеснение задач) должны быть предсказуемыми

Что делать:

- Сбрасывать кэши при старте задач
- Размещать данные по фиксированным адресам
- Избегать сложного вычисления адресов

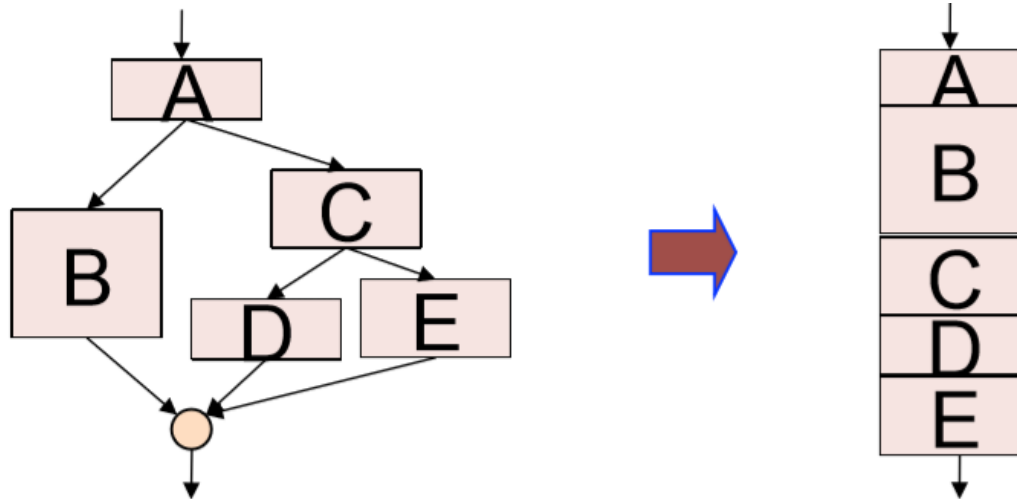
# Константные длительности операций процессора

- Все операции процессора должны быть реализованы так, чтобы выполняться за константное время, независимо от значений операндов (пример: целочисленный сумматор всегда отрабатывает за 1 такт)
- В частности, предикатные команды должны выполняться за константное время. Если предикат ложен, команда выполняется, но её результаты отбрасываются и не изменяют состояние процессора.
- У циклически выполняемых команд в коде должны быть константные значения числа итераций.

# Производительность линейного кода

Времена выполнения ветвей, выбираемых в зависимости от входных данных, суммируются в связи с линеаризацией.

- Время выполнения линейного кода довольно велико (по сравнению с оригинальным), если путь выполнения оригинального кода существенно зависит от входных данных



# Производительность линейного кода (2)

Процессоры с длинными конвейерами тратят много тактов на перезаполнение конвейера при ошибочном предсказании ветвления.

- Предикатное выполнение может быть «дешевле» ветвления
- Современные компиляторы и процессоры могут использовать предикатное выполнение для улучшения производительности

# Пример: ускорение за счет if-преобразования

```
if rA < rB then swap(rA, rB);
```

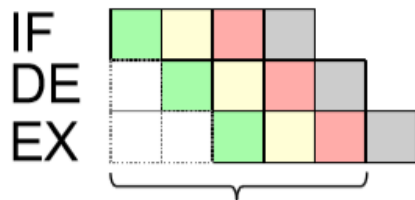
## Ветвящийся код

■ `cmplt rA, rB`  
■ `bf skip`  
■ `swp rA, rB`  
■ `skip:`

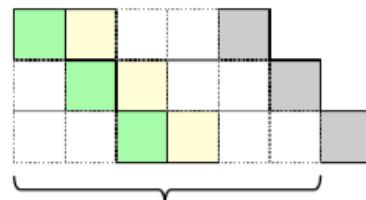
## Предикатный код

■ `predlt Pi, rA, rB`  
■ `(Pi) swp rA, rB`  
■

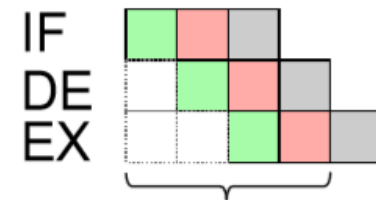
Выполнение в трёхступенчатом конвейере:



5 тактов



6 тактов

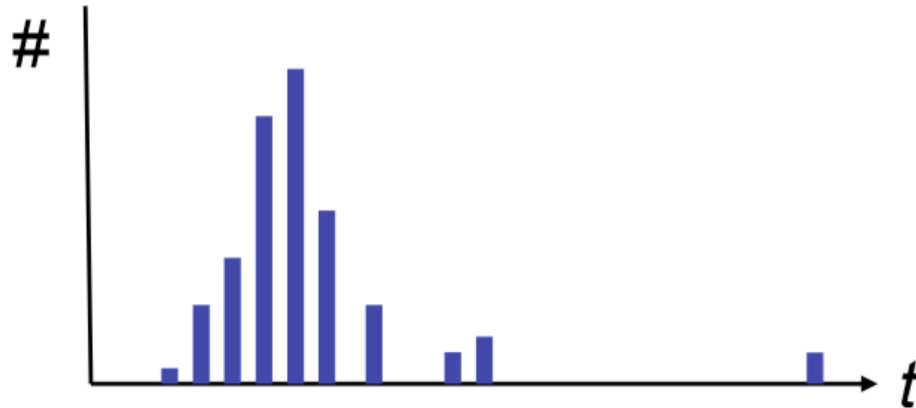


4 такта

# Свойства процедуры линеаризации кода

- **Полнота**: любой участок кода с ограниченным WCET может быть линеаризован
- В линеаризованном коде путь выполнения – **единственный**
- Анализ WCET **тривиален**: запустить код и измерить время (при «наихудшем» начальном состоянии системы)
- Код выполняется заметно дольше ветвящегося кода

# Времена выполнения

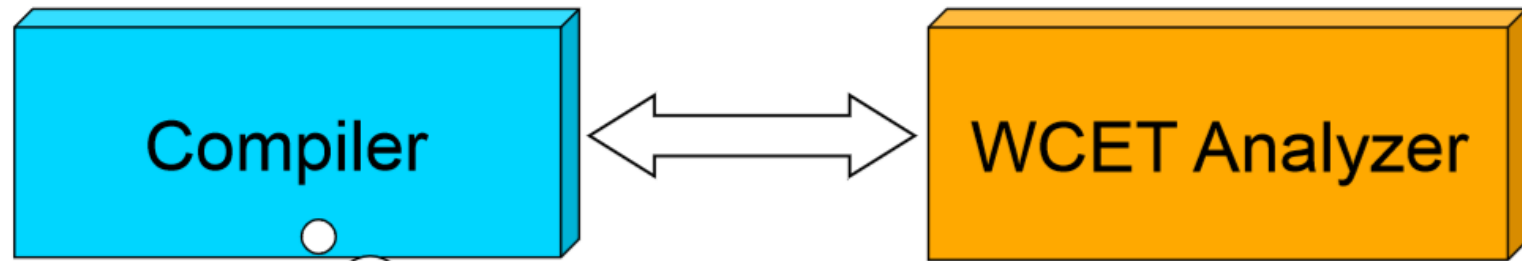


Времена выполнения  
до и после  
линеаризации



# Оптимизация WCET на этапе компиляции

# WCET-aware Code Optimizations

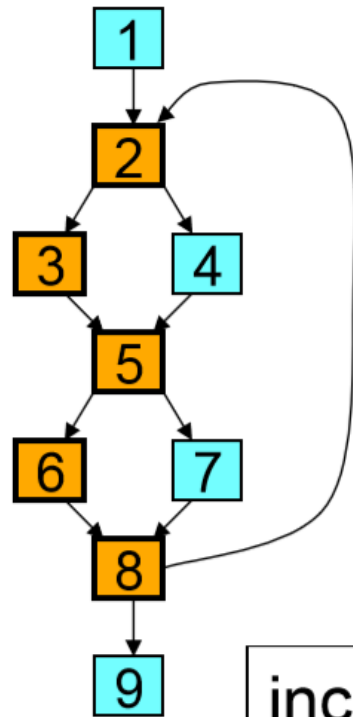


- localize worst-case path,
- select appropriate optimizations,
- identify code locations to apply optimizations

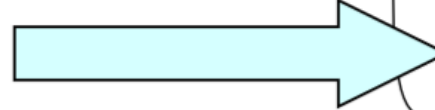
# WCET-aware Code Optimizations

## Superblock Formation:

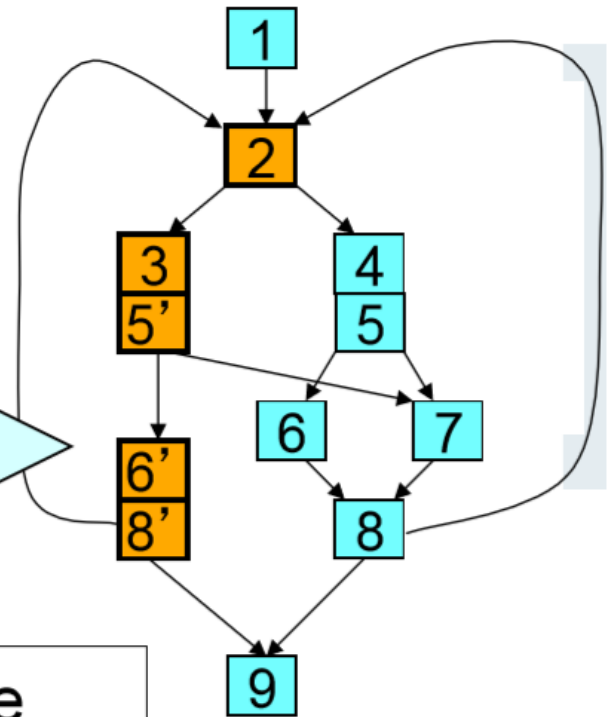
superblock:  
...block with  
single entry  
point



superblock  
formation



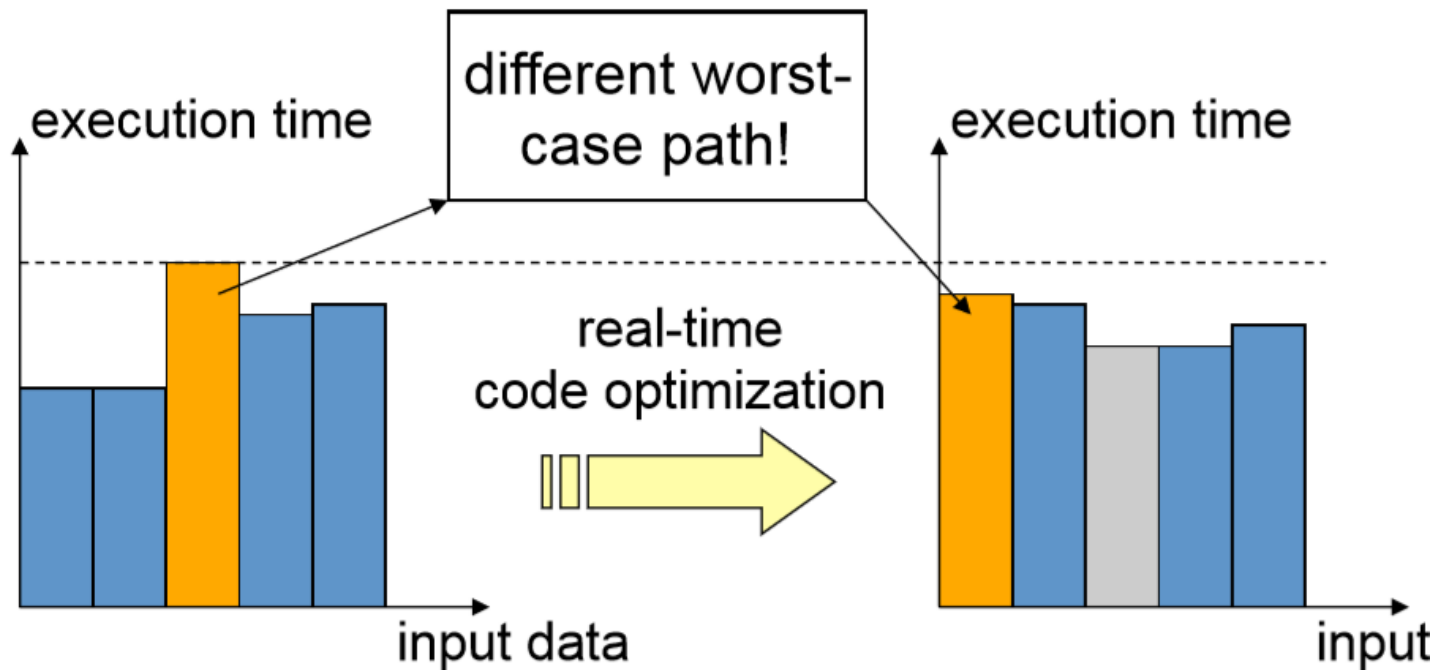
 ...worst case path



increase of code size  
but reduction of WCET

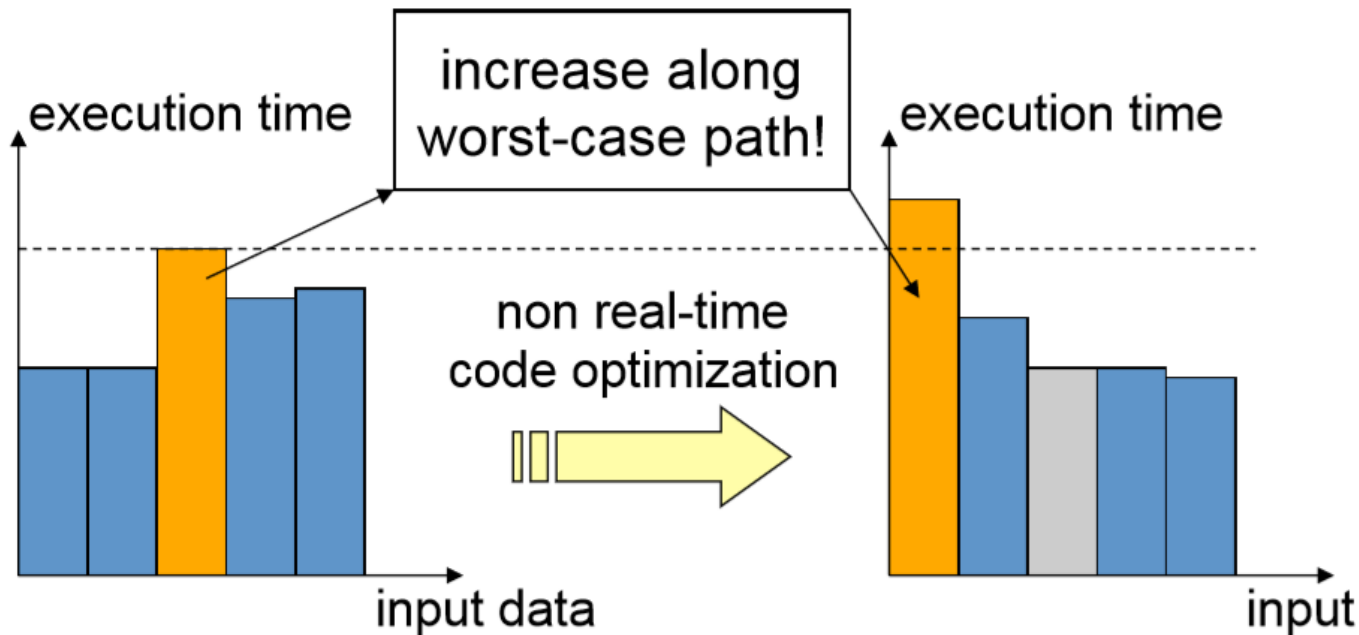
# Наихудший путь

- *Внимание:* в результате оптимизации другой путь может стать наихудшим



# Наихудший путь

- *Внимание:* новый наихудший путь может выполняться дольше, чем первоначальный



Или всё-таки замерять?

# Почему нельзя просто измерить WCET?

- ❑ Замер времени выполнения на **всех** путях выполнения реалистичной программы – на практике **невозможен**
- ❑ При определении тестовой выборки могут быть **упущены редкие сценарии** выполнения (обработка ошибочных ситуаций и т.п.)
- ❑ Выбранные тестовые данные **могут не породить самую длинную** (по времени) **трассу** выполнения
- ❑ Внутреннее **состояние процессора** на момент старта измерений может не быть наихудшим

Простые замеры могут послужить источником **первоначальной (грубой)** *нижней* оценки WCET

# С другой стороны...

Не во всех случаях строго необходима безопасная (не заниженная) оценка WCET

- Системы мягкого реального времени (например, мультимедиа)
- Системы, устойчивые к редким превышениям директивных сроков

Для новой аппаратной платформы быстрее всего можно начать именно замеры времени выполнения (а статические методы анализа аппаратных задержек адаптировать долго)

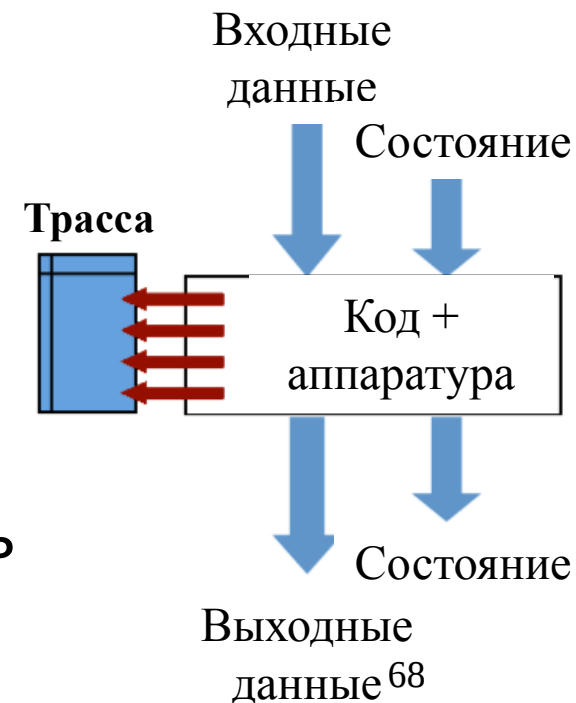
Низкие затраты на аннотирование кода => быстрое получение грубой оценки времени выполнения

Замеры дополняют статический анализ, предоставляя реальные значения задержек  
Замеры могут использоваться для уточнения результатов статического анализа WCET

- Моделировать некоторые современные процессоры *действительно* сложно
- Для некоторых задач зависимость времени выполнения от входных данных *по-настоящему* сложна (вещественные числа, массивы и т.п.)
- Люди из промышленности требуют подтверждения теоретических оценок WCET данными «из жизни»

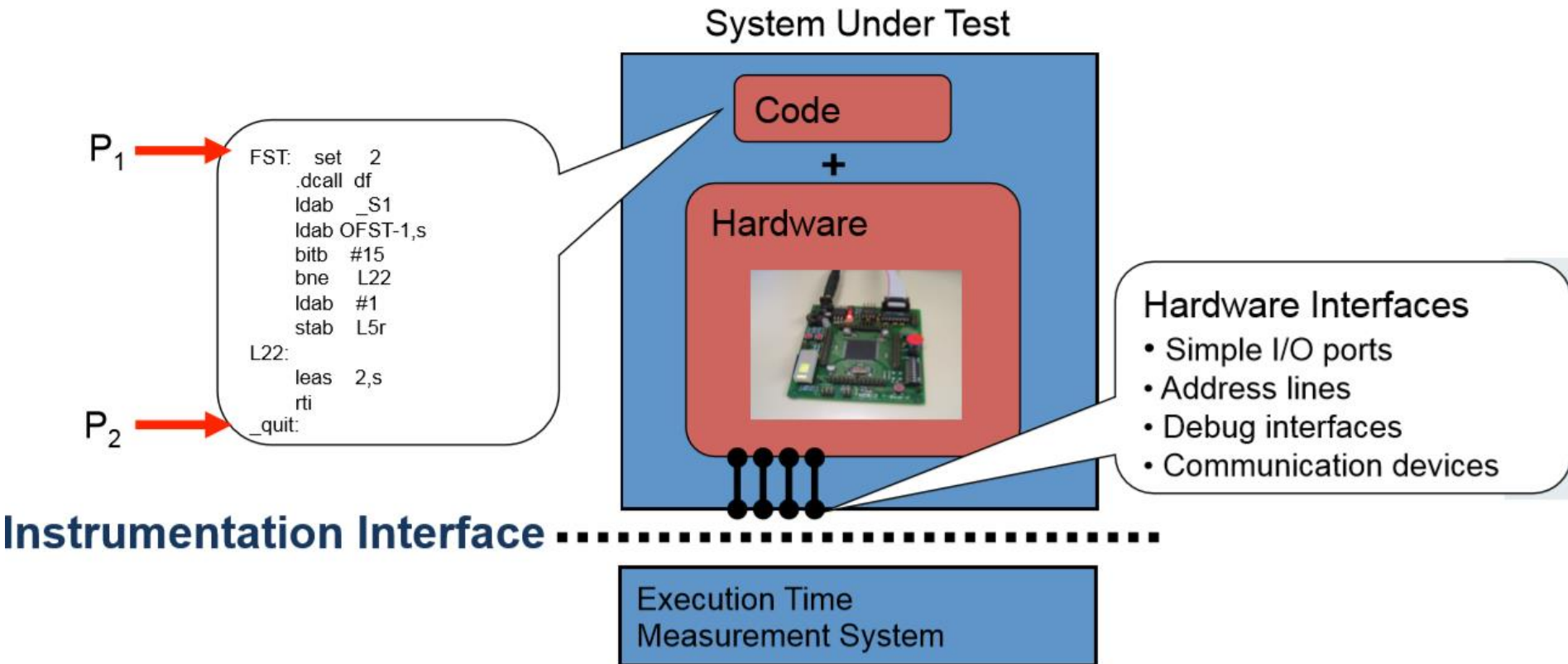
# Оценка WCET с помощью измерений

- Информация о таймингах получается путём **измерения** времени **выполнения** кода на реальной **целевой аппаратуре**
- Точки инструментирования кода формируют **наблюдаемые извне события**, используемые для старта и завершения измерений
- Трасса выполнения содержит собираемую *совместно* информацию о **путях** в программе и **временах** их **выполнения** (путь = последовательность линейных участков)



# Execution Time Measurements

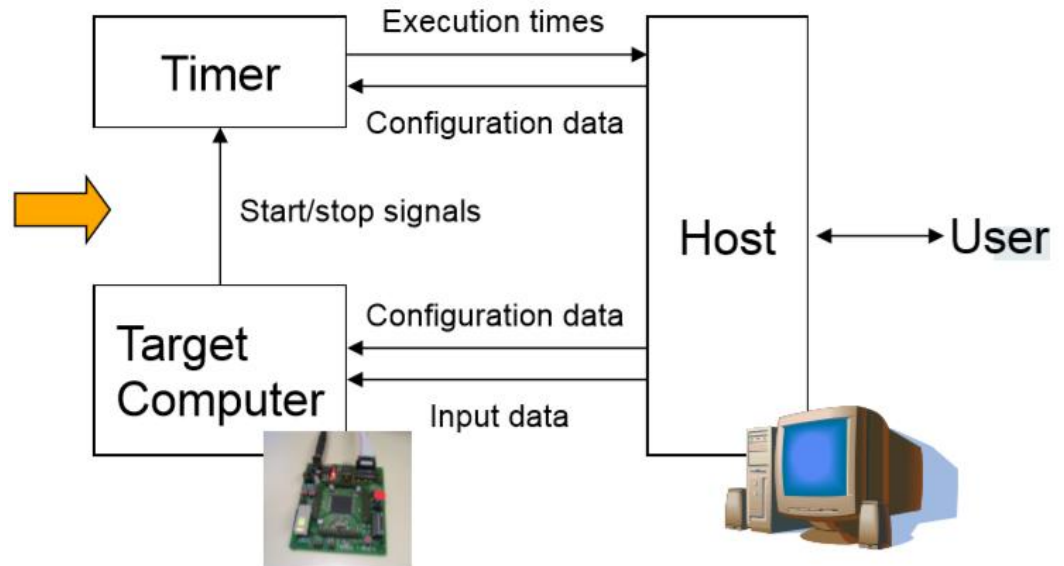
Goal: Obtain execution time for a **path**



- Необходимо СБРОСИТЬ состояние процессора (кэш, конвейер и т.п.) перед каждым прогоном
- НО: для измерения WCET конкретного пути необходимо специфическое начальное состояние кэша (например, если путь входит в состав второй или более поздней итерации цикла, команды должны *находиться в кэше*)

# Методы инструментирования

- Чисто аппаратное инструментирование
- Внешние замеры времени выполнения при помощи программно формируемых сигналов вовне
- Чисто программное (внутреннее) инструментирование



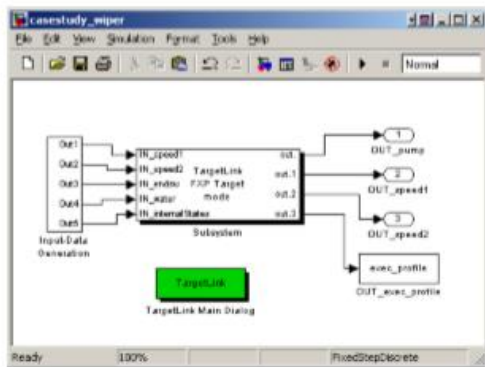
# Важные соображения по замерам

Как измерить именно то, что нужно:

- Инструментирование кода не должно изменять путь выполнения или время выполнения программы *непредсказуемым* или *неизвестным* способом. Точки инструментирования должны располагаться в фиксированных местах кода.
- Нужна уверенность в том, что запуски для замеров всегда стартуют с известного требуемого состояния процессора (кэш, конвейер, предсказание ветвлений и т.п.)

# Промышленный подход

- Пример промышленного процесса разработки:

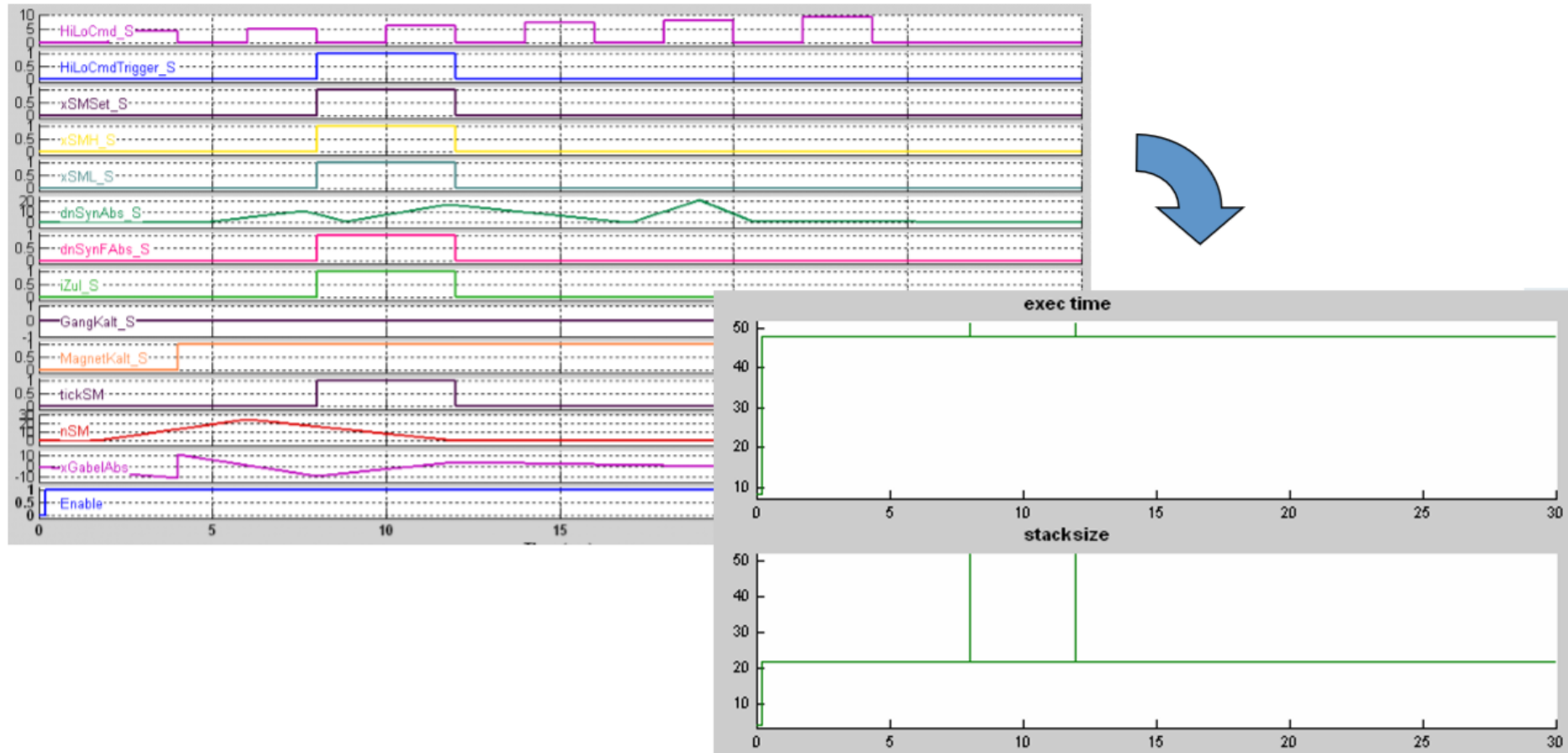


Проектирование  
Matlab + Simulink  
Matlab + RTW

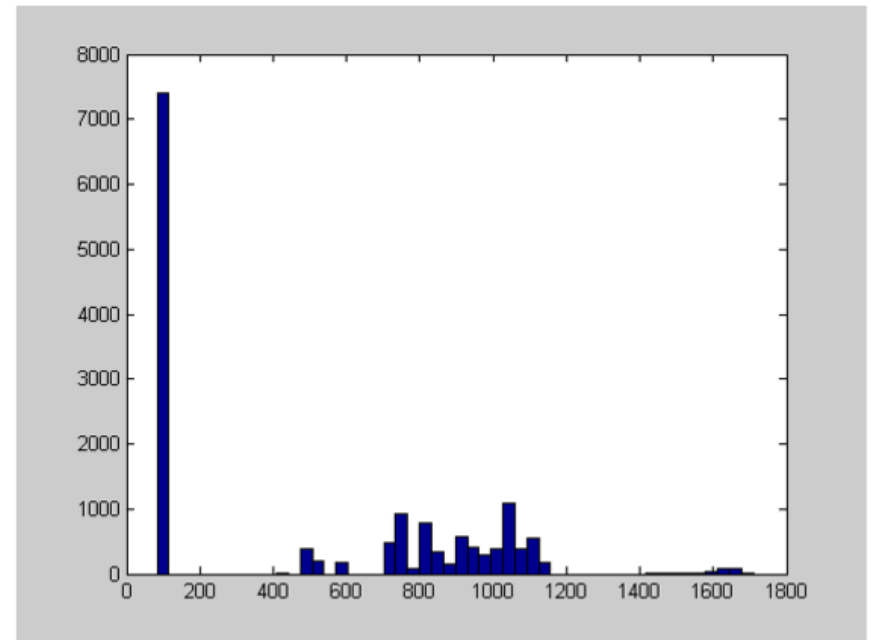
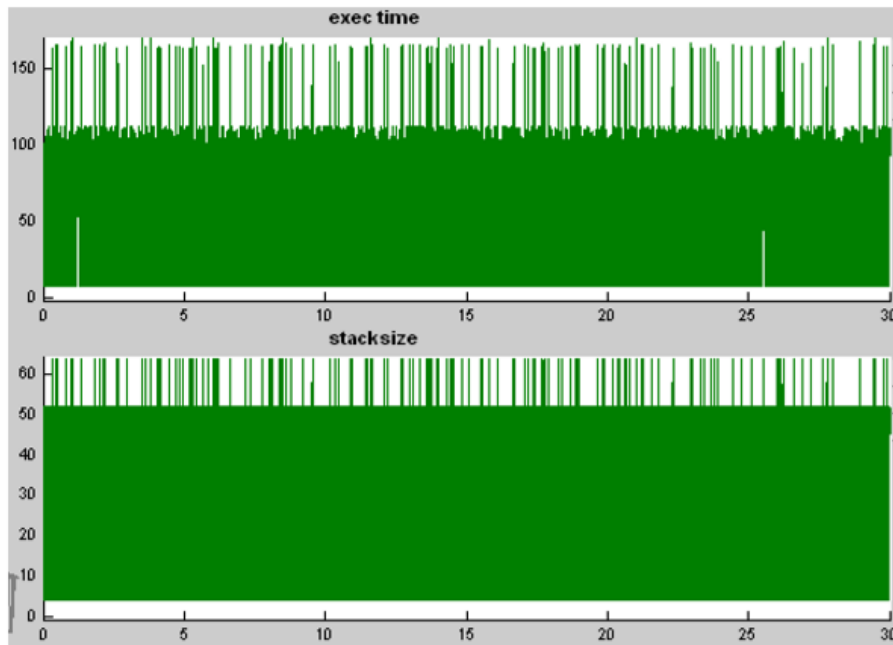
Тестирование  
Платы-прототипы  
Аппаратура целевой системы

Внедрение  
Аппаратура целевой  
системы

# Промышленный подход – массивы входных данных



# Промышленный подход – случайные данные

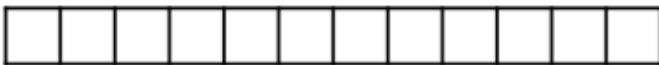


# Поиск «наихудших» входных данных – оптимизационная задача

- Массивы входных данных → ограниченный перебор
- Случайные входные данные → случайный поиск
- Как насчёт «умной» оптимизации?

# Evolutionary Algorithms (EA)

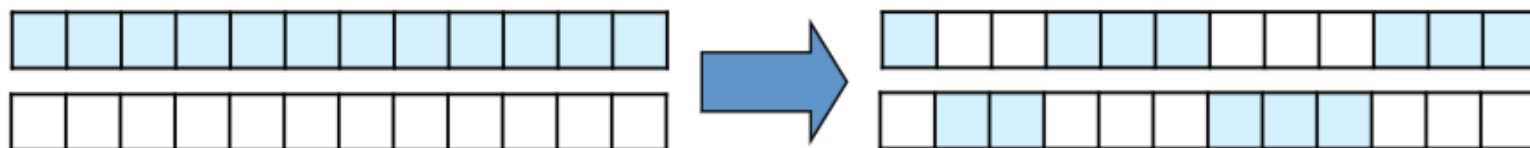
**Gene** = an independent property of an individual

**Individual** = vector of genes 

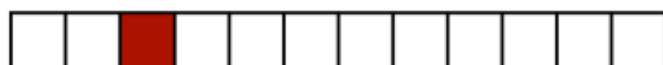
**Population** = set of a number individuals

**Fitness value** = chance of survival of an individual

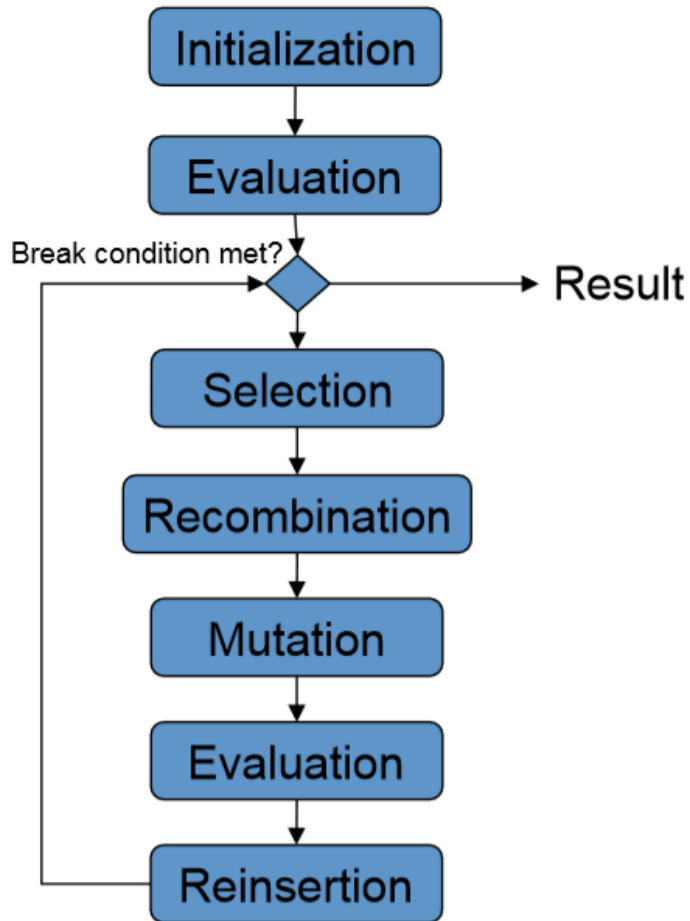
**Recombination** = mating of two individuals,  
exchange of genes



**Mutation** = random change of a gene



# Схема эволюционного алгоритма



- Отбор: выживание наиболее приспособленных. Отбор рандомизирован и выполняется с учётом значения целевой функции.
- Скрещивание: обмен генами, например одно- или n-точечное скрещивание
- Мутация: рандомизированное изменение генов

# Оценка WCET эволюционными алгоритмами

- Ген = значение входной или внутренней переменной
- Целевая функция = измеренное время выполнения (больше значение => особь лучше)
- Результат = «наилучшая особь» = особь (набор данных) с наибольшим временем выполнения программы
- Даёт хорошую, но *небезопасную* оценку WCET

# Оценка WCET эволюционными алгоритмами

```
if(x) {  
    fast();  
} else {  
    slow();  
}  
if(y) {  
    slow();  
} else {  
    fast();  
}
```

- Старт

- [0]  $x = 0, y = 0 \rightarrow ET: 40$

- [1]  $x = 1, y = 1 \rightarrow ET: 40$

- Скрещивание:

- [2]  $x = 0, y = 1 \rightarrow ET: 50$

- [3]  $x = 1, y = 0 \rightarrow ET: 30$

Алгоритм завершается, если значение целевой функции не улучшается на протяжении заданного числа итераций

# Результат применения эволюционного алгоритма

| Программа | WCET       | WCET SA    | WCET EA    |                          |
|-----------|------------|------------|------------|--------------------------|
| Matrix    | 13,190,619 | 15,357,471 | 13,007,019 | Завышенная<br>Заниженная |
| Sort      | 11,872,718 | 24,469,014 | 11,826,117 |                          |
| Graphics  | N/A        | 2,602      | 2,176      |                          |
| Railroad  | N/A        | 23,466     | 22,626     |                          |
| Defense   | N/A        | 72,350     | 35,226     |                          |

SA .... Static analysis  
EA .... Evolutionary algorithms

Нет результата

Точность - ?

[Mueller, Wegener, RTSS1998]

# Выводы

- Оценка WCET – сложная задача, разрешимая (безопасно и с приемлемой точностью) только для некоторых типов *процессоров* и *программ*
- На практике для оценки WCET применяется сочетание формальных методов и измерений
- Система реального времени должна быть устойчива к отдельным/локальным превышениям имеющихся оценок WCET
  - сторожевые таймеры для обнаружения превышения WCET
  - устойчивость к сбросу задачи-нарушителя (и пропуску итерации её выполнения)
  - «растяжимые» периоды выполнения задач для динамической регулировки загрузки процессора

**Спасибо за внимание!**