Нижегородский государственный университет
им. Н.И.Лобачевского

Факультет Вычислительной математики и кибернетики

Параллелизм как основа архитектуры ВС

Раздел 4

Статическая конвейеризация

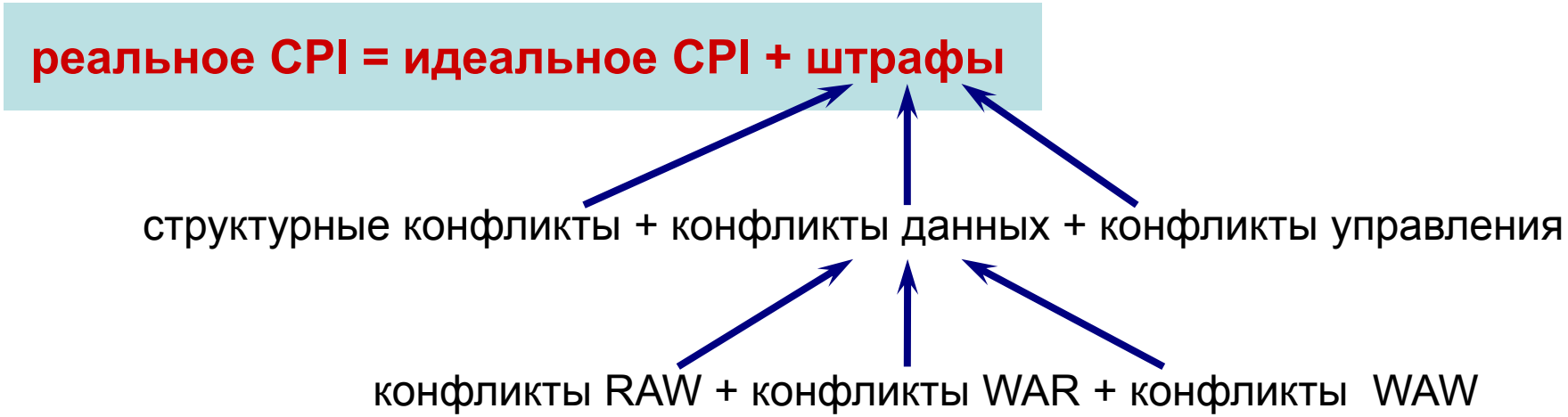
Кудин А.В., к.т.н.

Содержание

- Возможности оптимизирующей компиляции на уровне инструкций
- Риски, метрики эффективности, оптимизация эффективности продвижения
- Методы снижения издержек и повышения производительности
- Классификация зависимостей между инструкциями
- Символическое разворачивание циклов

Параллелизм уровня инструкций (ILP)

Параллелизм уровня инструкций (Instruction Level Parallelism) – исполнение последовательности независимых инструкций параллельно (с перекрытием). Конвейерная обработка повышает производительность именно за счёт перекрытия независимых инструкций.



Программы, обладающие большим ILP кода (имеющие меньше зависимостей), обычно показывают лучшую производительность на CPU с конвейером.

Анализ ILP

Пример 1:

```
ADD.D   F2, F4, F6
ADD.D   F10, F4, F8
ADD.D   F12, F12, F6
```

Инструкции независимы!
Могут исполняться параллельно.
Могут исполняться в любом порядке!

Такая последовательность кода имеет **высокую степень ILP**.

Пример 2:

```
ADD.D   F2, F4, F6
ADD.D   F10, F8, F2
ADD.D   F12, F2, F10
```

Инструкции зависимы…
Они не могут исполняться параллельно…

Такая последовательность кода имеет **низкую степень ILP**.

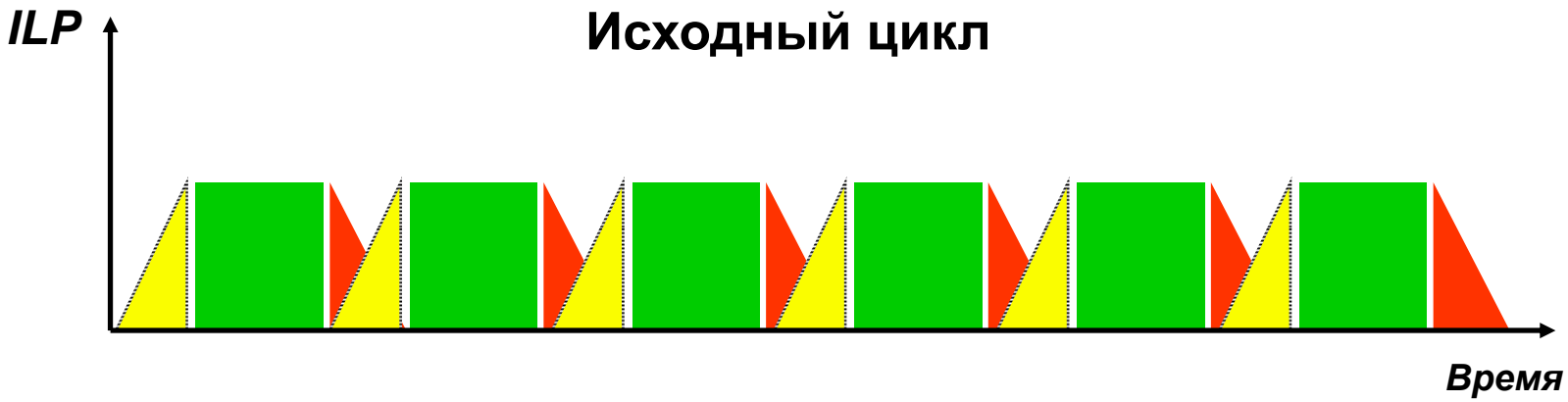
Базовый блок инструкций (ББИ)

Базовый блок инструкций – линейная последовательность кода без переходов внутрь или наружу (например, тело простого цикла).

Параллелизм на уровне инструкций в ББИ ограничен размером ББИ и имеющимися зависимостями между инструкциями ББИ. В типичном коде из целочисленных операций частота переходов составляет около 15 процентов. Таким образом, типичный размер ББИ около 7 инструкций. Однако время разгона современных гиперконвейеров намного больше 7 тактов.

Любой статический метод, увеличивающий средний размер ББИ, увеличивает и ILP кода, так как больший ББИ даёт больше возможностей для оптимального статического планирования конвейера компилятором.

Программная конвейеризация циклов



Параллелизм уровня цикла (LLP)

Согласно принципу локальности типичного кода, превалирующая доля времени исполнения программ приходится на циклы.

Стандартный способ увеличения ILP состоит в использовании параллелизма между итерациями цикла – **параллелизма уровня цикла** (Loop Level Parallelism).

Увеличение ILP достигается путём разворачивания циклов либо статически компилятором (статическое планирование), либо динамически процессором (динамическое планирование). Такое разворачивание циклов увеличивает размер ББИ, что позволяет удалять больше тактов простоя, так как больше инструкций могут быть переупорядочены (reordered) либо статически компилятором, либо динамически процессором.

Пример разворачивания циклов

Исходный цикл на Си

```
for (i=1000; i>0; i--)
  x[i] = x[i] + s;
```

итерации независимы

Код MIPS

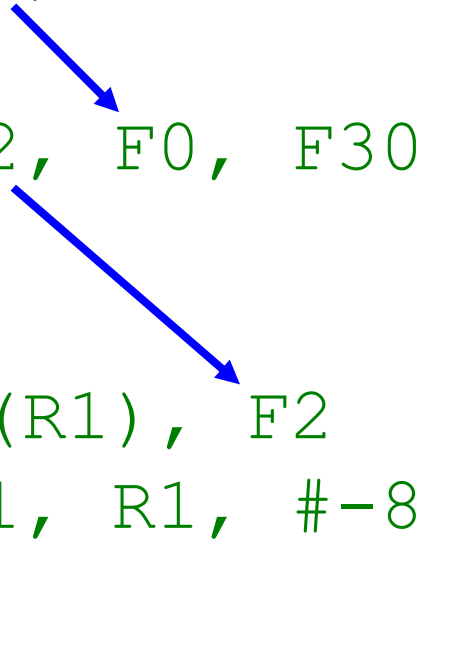
```
Loop: L.D    F0, 0(R1)
      ADD.D  F2, F0, F30
      S.D    0(R1), F2
      DADDUI R1, R1, #-8
      BNE    R1, R2, Loop
```

размер ББИ = 5

Пример разворачивания циклов

Исходный код MIPS

```
Loop: L.D     F0, 0(R1)
      простой
      ADD.D   F2, F0, F30
      простой
      простой
      S.D     0(R1), F2
      DADDUI  R1, R1, #-8
      простой
      BNE     R1, R2, Loop
      простой
```



для пятиступенчатого конвейера MIPS:
10 тактов на итерацию
CPI = 2

Пример разворачивания циклов

Развёрнутый цикл, фактор 4

```
Loop: L.D    F0, 0(R1)
      ADD.D  F2, F0, F30
      S.D    0(R1), F2
      L.D    F0, -8(R1)
      ADD.D  F2, F0, F30
      S.D    -8(R1), F2
      L.D    F0, -16(R1)
      ADD.D  F2, F0, F30
      S.D    -16(R1), F2
      L.D    F0, -24(R1)
      ADD.D  F2, F0, F30
      S.D    -24(R1), F2
      DADDUI R1, R1, #-32
      BNE    R1, R2, Loop
```

размер ББИ = 14

накладной расход
на организацию цикла
снижен в 4 раза

зависимости и простои
пока остаются

Пример разворачивания циклов

Развёрнутый цикл, фактор 4, зависимости по именам устранены

```
Loop: L.D    F0, 0(R1)
      ADD.D  F2, F0, F30
      S.D    0(R1), F2
      L.D    F4, -8(R1)
      ADD.D  F6, F4, F30
      S.D    -8(R1), F6
      L.D    F8, -16(R1)
      ADD.D  F10, F8, F30
      S.D    -16(R1), F10
      L.D    F12, -24(R1)
      ADD.D  F14, F12, F30
      S.D    -24(R1), F14
      DADDUI R1, R1, #-32
      BNE    R1, R2, Loop
```

цикл без повторного использования регистров устраняет зависимости WAW и WAR

Пример разворачивания циклов

Перепланированный развёрнутый цикл, фактор 4

```
Loop: L.D    F0, 0(R1)
      L.D    F4, -8(R1)
      L.D    F8, -16(R1)
      L.D    F12, -24(R1)
      ADD.D  F2, F0, F30
      ADD.D  F6, F4, F30
      ADD.D  F10, F8, F30
      ADD.D  F14, F12, F30
      S.D    0(R1), F2
      S.D    -8(R1), F6
      S.D    -16(R1), F10
      S.D    -24(R1), F14
      DADDUI R1, R1, #-32
      BNE    R1, R2, Loop
```

зависимости и простои устранены

для пятиступенчатого конвейера MIPS:
3.5 такта на исходную итерацию

фактор развёртки ограничен размером регистрового файла

Символическое разворачивание циклов

Компоненты повышения производительности:

- Больший размер ББИ – больше комбинаторных возможностей для перепланирования инструкций (удаление большего числа простоев)
- Исполняется меньшее количество инструкций поддержки циклов

Необходимые условия:

- Независимость итераций цикла
- Большое количество регистров для предотвращения конфликтов по именам регистров (характерно для RISC)

Пример программной конвейеризации циклов

Если размер регистрового файла недостаточен для проведения символического разворачивания цикла (максимальный фактор равен единице), можно воспользоваться методом программной конвейеризации циклов

Исходный код MIPS

```
Loop: L.D    F0, 0(R1)
      ADD.D  F2, F0, F30
      S.D    0(R1), F2
      DADDUI R1, R1, #-8
      BNE    R1, R2, Loop
```

цепочка зависимостей в теле исходного цикла состоит из трёх инструкций: **L.D → ADD.D → S.D**

Пример программной конвейеризации циклов

Развернём цикл три раза
(так как длина цепочки равна трём)
без переименования регистров

```
Loop: L.D     F0, 0(R1)
      ADD.D   F2, F0, F30
      S.D     0(R1), F2
      L.D     F0, -8(R1)
      ADD.D   F2, F0, F30
      S.D     -8(R1), F2
      L.D     F0, -16(R1)
      ADD.D   F2, F0, F30
      S.D     -16(R1), F2
      DADDUI  R1, R1, #-24
      BNE     R1, R2, Loop
```

итерация x [i]

итерация x [i-1]

итерация x [i-2]

Пример программной конвейеризации циклов

Развернём цикл три раза
(так как длина цепочки равна трём)
без переименования регистров

```
Loop: L.D     F0, 0(R1)
      ADD.D   F2, F0, F30
      S.D     0(R1), F2
      L.D     F0, -8(R1)
      ADD.D   F2, F0, F30
      S.D     -8(R1), F2
      L.D     F0, -16(R1)
      ADD.D   F2, F0, F30
      S.D     -16(R1), F2
      DADDUI  R1, R1, #-24
      BNE     R1, R2, Loop
```

Программно конвейеризованный цикл

```
      L.D     F0, 0(R1)
      ADD.D   F2, F0, F30
      L.D     F0, -8(R1)
Loop: S.D     0(R1), F2
      ADD.D   F2, F0, F30
      L.D     F0, -16(R1)
      DADDUI  R1, R1, #-8
      BNE     R1, R2, Loop
      S.D     0(R1), F2
      ADD.D   F2, F0, F30
      S.D     -8(R1), F2
```

Пример программной конвейеризации циклов

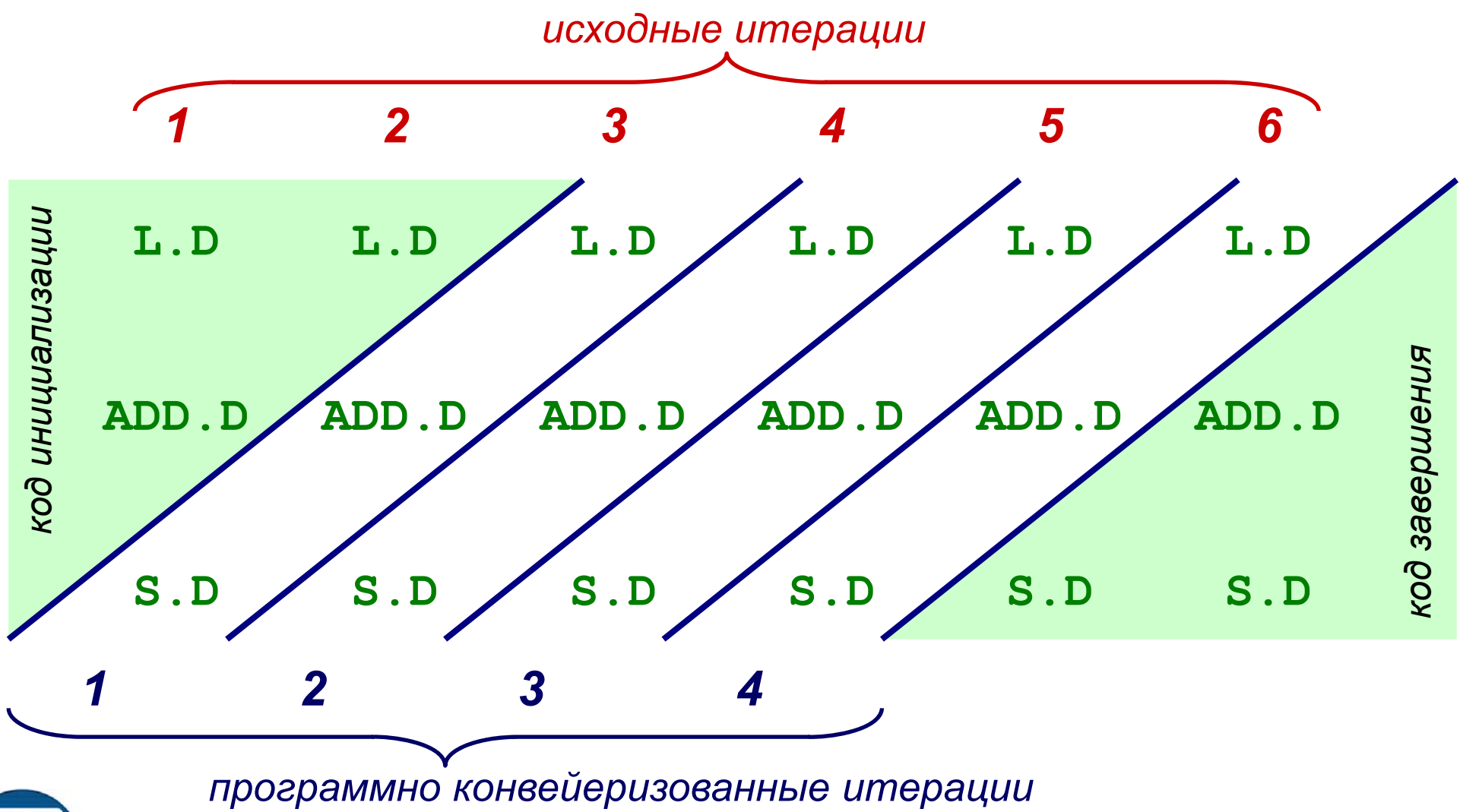
Программно конвейеризованный цикл

```
        L.D     F0, 0(R1)
        ADD.D   F2, F0, F30
        L.D     F0, -8(R1)
Loop:   S.D     0(R1), F2     ; запись x[i]
        ADD.D   F2, F0, F30   ; модификация x[i-1]
        L.D     F0, -16(R1)   ; загрузка x[i-2]
        DADDUI  R1, R1, #-8
        BNE     R1, R2, Loop
        S.D     0(R1), F2
        ADD.D   F2, F0, F30
        S.D     -8(R1), F2
```

код инициализации (остатки первых итераций исходного цикла)

код завершения (остатки последних итераций исходного цикла)

Пример программной конвейеризации циклов



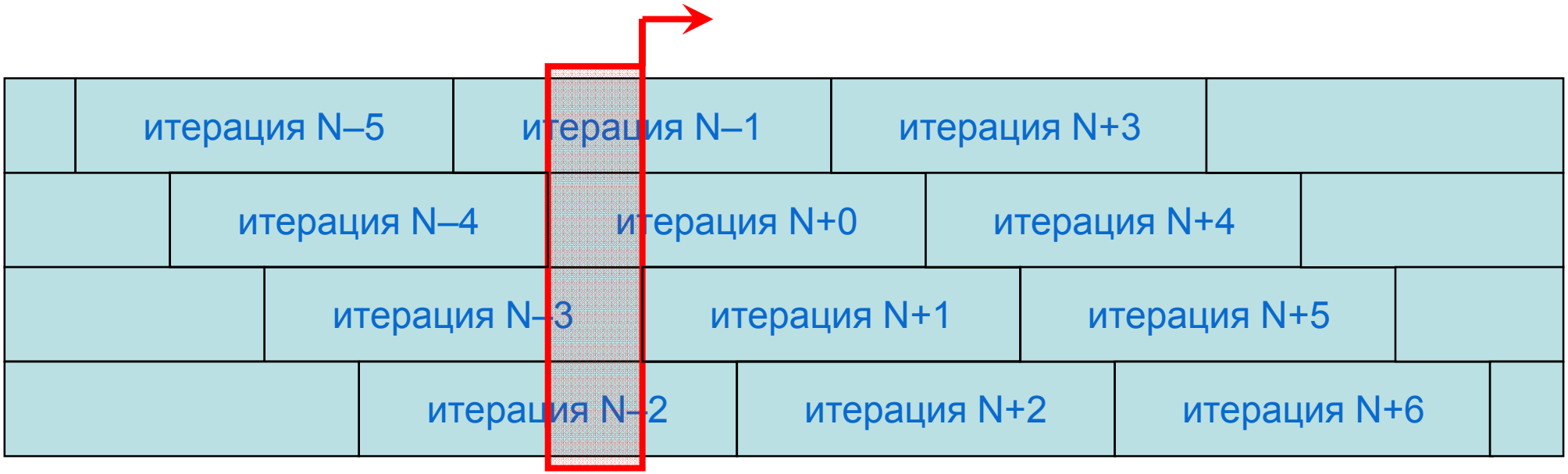
Программная конвейеризация циклов

- При программной конвейеризации циклов внутриитерационные зависимости преобразуются в межитерационные
- Программная конвейеризация циклов оптимизирующими компиляторами в статическом режиме аналогична аппаратному переупорядочиванию инструкций процессорами в динамическом режиме (см. раздел “Динамическое планирование”)

Сравнение методов конвейеризации

	статическая конвейеризация	динамическая конвейеризация
	оптимизирующий компилятор или программист	ядро CPU с неупорядоченной моделью обработки
имеющиеся ресурсы	большая память, много времени	минимальны
состояние вычислительного процесса	не определено	определено

Конвейеризация циклов

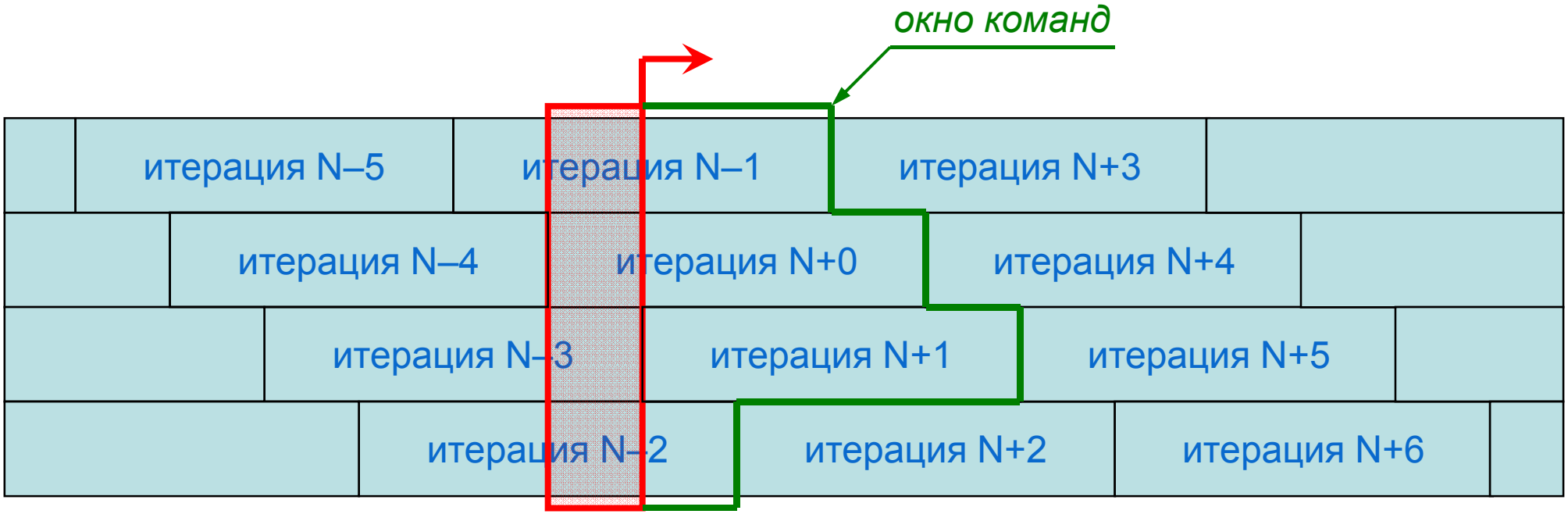


программно конвейеризованная итерация

либо

инструкции, избранные динамическим планировщиком

Конвейеризация циклов



программно конвейеризованная итерация

либо

инструкции, избранные динамическим планировщиком

Анализ параллелизма уровня цикла

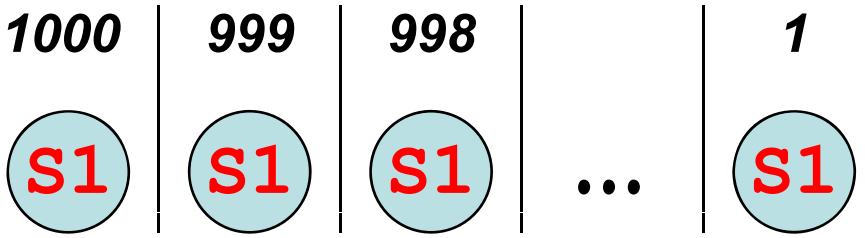
Анализ LLP состоит в выявлении фактов зависимости данных на более поздних итерациях цикла от значений, полученных на более ранних итерациях, и возможности превращения итераций цикла в независимые (параллельные).

Знакомый пример

```
for (i=1000; i>0; i--)
  x[i] = x[i] + s;      // S1
```

*Все итерации (**S1**) независимы!*

Граф зависимостей:

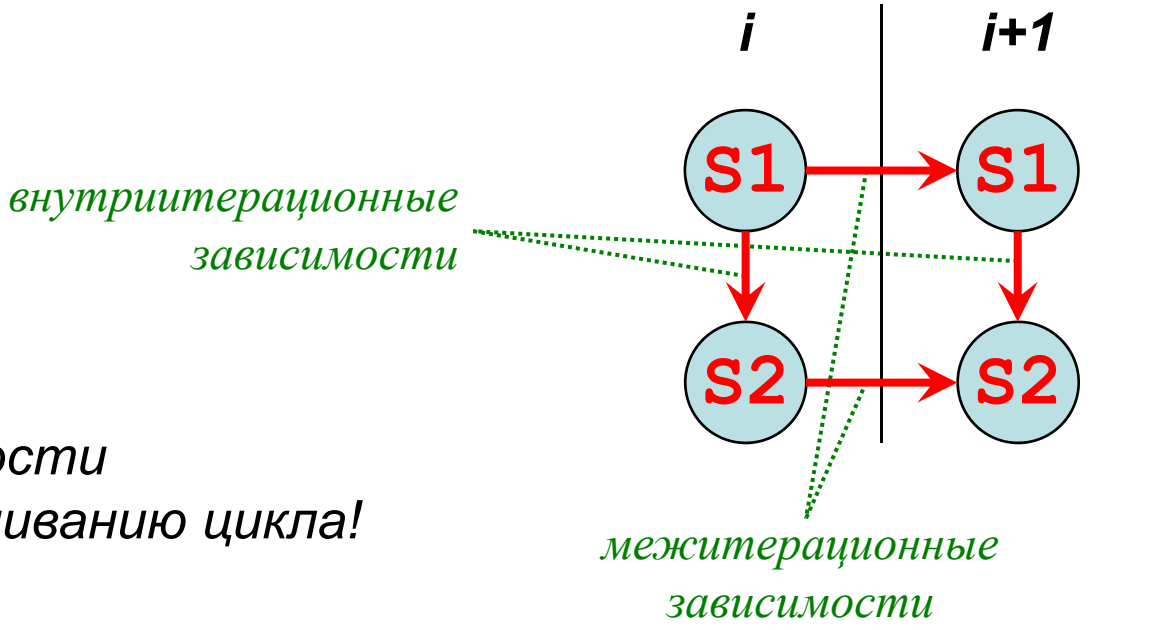


Пример анализа LLP

```
for (i=1; i<=100; i++) {
    A[i+1] = A[i] + C[i];    // S1
    B[i+1] = B[i] + A[i+1]; // S2
}
```

пусть A,B,C – массивы без перекрытий

Граф зависимостей:



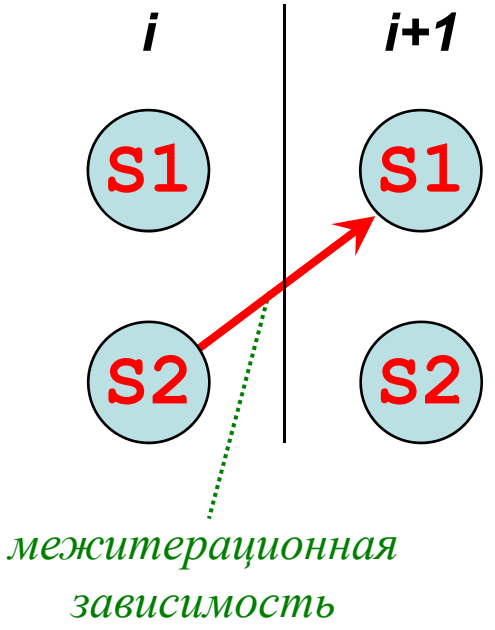
Межитерационные зависимости препятствуют распараллеливанию цикла!

Пример анализа LLP

```
for (i=1; i<=100; i++) {
    A[i] = A[i] + B[i];    // S1
    B[i+1] = C[i] + D[i]; // S2
}
```

пусть A,B,C,D – массивы без перекрытий

Граф зависимостей:



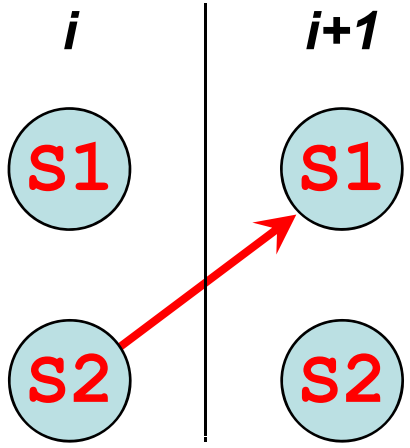
Превратим межитерационную зависимость во внутриитерационную?

Пример анализа LLP

исходный код

```
for (i=1; i<=100; i++) {
   A[i] = A[i] + B[i];
   B[i+1] = C[i] + D[i];
}
```

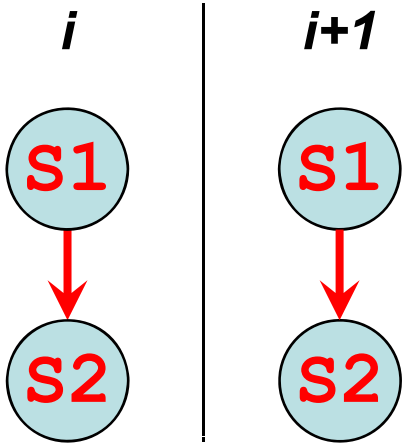
Граф зависимостей:



оптимизированный код

```
A[1] = A[1] + B[1];
for (i=1; i<=99; i++) {
   B[i+1] = C[i] + D[i];
   A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

Граф зависимостей:



Тест GCD

Для обнаружения межитерационных зависимостей компилятор может использовать **тест GCD**.

Пусть в итерации i_1 элемент массива с индексом $[a \times i_1 + b]$ сохраняется, а в более поздней итерации i_2 этот же элемент массива с индексом $[c \times i_2 + d]$ загружается.
Тогда межитерационная зависимость существует, если $(d - b)$ делится на **НОД(c,a)**.

Теста GCD (Greatest Common Divisor) достаточно, чтобы гарантировать отсутствие межитерационной зависимости.

Тест не учитывает границы цикла. Так, например, при однократном исполнении тела цикла никаких межитерационных зависимостей быть не может (короткие циклы обычно линеаризуются).

Вопросы для обсуждения

- Каковы отличия между параллелизмом уровня команд и микроуровневым параллелизмом?
- Какими факторами (в случае “идеальной” последовательности инструкций) ограничена степень ILP в процессорах различных архитектур?
- Почему для наибольшего увеличения ILP необходимо одновременное применение статической конвейеризации и динамического планирования?
- Почему превалирующее внимание при проведении статической конвейеризации сосредоточено на оптимизации циклов?
- Каковы преимущества и недостатки символического разворачивания циклов против программной конвейеризации циклов?
- В каких случаях межитерационные зависимости не препятствуют распараллеливанию циклов?

Следующая тема

- **Динамическое планирование**