8.1 Эффективность конвейера

В предыдущей лекции мы рассмотрели средства конвейеризации, которые обеспечивают совмещенный режим выполнения команд, когда команды являются независимыми друг от друга. Это потенциальное совмещение выполнения команд называется *параллелизмом на уровне команд.*

Сейчас мы рассмотрим ряд методов развития идей конвейеризации, основанных на увеличении **степени параллелизма,** используемой при выполнении команд и для начала познакомимся с методами, позволяющими снизить влияние конфликтов по данным и по управлению.

И так… Среднее количество тактов (CPI) для выполнения команды в конвейере можно определить как:

CPI конвейера = CPI идеального конвейера
+ Приостановки из-за структурных конфликтов
+ Приостановки из-за конфликтов типа RAW
+ Приостановки из-за конфликтов типа WAR
+ Приостановки из-за конфликтов типа WAW
+ Приостановки из-за конфликтов по управлению

CPI идеального конвейера есть не что иное, как максимальная пропускная способность, достижимая при реализации.

Уменьшая каждое из слагаемых в правой части выражения, минимизируем общий CPI конвейера и таким образом увеличиваем пропускную способность команд. Это выражение позволяет также охарактеризовать различные методы сокращения CPI, по тому компоненту общего CPI, который соответствующий метод уменьшает.

Укажем некоторые методы, которые рассмотрим далее, и их воздействие на величину CPI (Таб. 8.1.1).

Таблица 8.1.1 Методы повышения эффективности конвейера

Метод	Снижает
Разворачивание циклов	Приостановки по управлению
Базовое планирование конвейера	Приостановки RAW
Динамической планирование с централизованной схемой управления	Приостановки RAW
Динамическое планирование с переименованием регистров	Приостановки WAR и WAW
Динамическое прогнозирование переходов	Приостановки по управлению
Выдача нескольких команд в одном такте	Идеальный CPI
Анализ зависимостей компилятором	Идеальный CPI и приостановки по данным
Программная конвейеризация и планирование трасс	Идеальный CPI и приостановки по данным
Выполнение по предположению	Все приостановки по данным и управлению
Динамическое устранение неоднозначности памяти	Приостановки RAW, связанные с памятью

8.2 Параллелизм уровня команд: зависимости и конфликты по данным

Все рассматриваемые методы используют параллелизм, заложенный в последовательности команд. Как было указано выше, этот тип параллелизма называется параллелизмом *уровня команд или **ILP.***

Рассмотрим **базовый** блок - линейную последовательность команд, переходы извне которой разрешены только на ее вход, а переходы внутри которой разрешены только на ее выход.

Степень параллелизма, доступная внутри базового блока достаточно мала. Например, средняя частота переходов в целочисленных программах составляет около 16%. Это означает, что в среднем между двумя переходами выполняются примерно пять команд. Поскольку эти пять команд возможно взаимозависимые, то степень перекрытия, которую можно использовать внутри базового блока, возможно, будет меньше чем пять. Чтобы *получить существенное улучшение производительности, необходимо использовать **параллелизм уровня команд одновременно для нескольких базовых блоков.***

Самый простой и общий способ увеличения степени параллелизма, доступного на уровне команд, является использование параллелизма между итерациями цикла. Этот тип параллелизма часто называется *параллелизмом **уровня итеративного цикла.***

Ниже приведен простой **пример цикла,** выполняющего сложение двух 1000-элементных векторов, который является полностью параллельным:

```
for (i = 1; i <= 1000; i = i + 1)

x[i] = x[i] + y[i];
```

Каждая итерация цикла может перекрываться с любой другой итерацией, хотя внутри каждой итерации цикла практическая возможность перекрытия небольшая.

Имеется несколько методов для ***превращения такого параллелизма уровня цикла в параллелизм уровня команд.*** Эти методы основаны главным образом на разворачивании цикла либо статически, используя компилятор, либо динамически с помощью аппаратуры. Ниже рассмотрим подробный пример разворачивания цикла.

Важным альтернативным методом использования параллелизма уровня команд является использование **векторных команд.** По существу векторная команда оперирует с последовательностью элементов данных. Например, приведенная выше последовательность на типичной **векторной машине** может быть выполнена с помощью четырех команд:

- двух команд загрузки векторов x и y из памяти,
- одной команды сложения двух векторов **и**
- одной команды записи вектора-результата.

Конечно, эти команды могут быть конвейеризованными и иметь относительно большие задержки выполнения, но эти задержки могут перекрываться.

8.2 Зависимости между командами

Чтобы точно определить, что понимается под параллелизмом уровня цикла и параллелизмом уровня команд, а также для количественного определения степени доступного параллелизма, необходимо определить, что такое параллельные команды и параллельные циклы.
Начнем с определения того, что такое пара параллельных команд.

*Две команды являются **параллельными,** если они могут выполняться в конвейере одновременно без приостановок, предполагая, что конвейер имеет достаточно ресурсов (структурные конфликты отсутствуют).*

Поэтому, если между двумя командами существует взаимозависимость, то они не

являются параллельными.

Имеется три типа зависимостей между командами:

- *зависимости по данным,*
- *зависимости по именам и*
- *зависимости по управлению.*

Зависимости по управлению сохраняются путем реализации схемы обнаружения конфликта по управлению, которая приводит к приостановке конвейера по управлению. Приостановки по управлению могут устраняться или уменьшаться множеством аппаратных и программных методов. Например, задержанные переходы могут уменьшать приостановки, возникающие в результате конфликтов по управлению. Другие методы уменьшения приостановок, вызванных конфликтами по управлению, включают разворачивание циклов, преобразование условных переходов в условно выполняемые команды и планирование по предположению, выполняемое с помощью компилятора или аппаратуры.

Далее рассмотрим некоторые из этих методов.

8.3 Параллелизм уровня цикла: концепции и методы

Цикл является параллельным, если отсутствует циклическая зависимость.

Параллелизм уровня цикла обычно анализируется на уровне исходного текста программы или близкого к нему, в то время как анализ параллелизма уровня команд главным образом выполняется, когда команды уже сгенерированы компилятором.

Анализ на уровне циклов включает определение того, какие зависимости существуют между операндами в цикле в пределах одной итерации цикла.

Будем рассматривать только зависимости по данным, которые возникают, когда операнд записывается в некоторой точке и считывается в некоторой более поздней точке. Мы обсудим коротко зависимости по именам. Анализ параллелизма уровня цикла фокусируется на определении

того, зависят ли по данным обращения к данным в последующей итерации от значений данных, вырабатываемых в более ранней итерации.

Рассмотрим различные варианты зависимостей по данным внутри циклов на примерах.

Пусть имеем следующий цикл:

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

Предположим, что A, B и C представляют собой отдельные, неперекрывающиеся массивы. Проанализируем, какие зависимости по данным имеют место между операторами этого цикла?

В этом цикле имеются две различных зависимости:

1. S1 использует значение, вычисляемое оператором S1 на более ранней итерации, поскольку итерация i вычисляет A[i+1], которое затем используется в итерации i+1. То же самое справедливо для оператора S2 для B[i] и B[i+l].

2. S2 использует значение A[i+1], вычисляемое оператором S1 в той же самой итерации.

Эти две зависимости отличаются друг от друга.

Предположим, что в каждый момент времени существует только одна из этих зависимостей. Рассмотрим зависимость оператора S1 от более ранней итерации S1. Эта зависимость *{loop-carried dependence)* означает, что между различными итерациями цикла существует зависимость по данным. Более того, поскольку оператор S1 зависит от самого себя, последовательные итерации оператора S1 должны выполняться упорядочено.

Вторая зависимость (S2 зависит от S1) не передается от итерации к итерации. Таким образом, если бы это была единственная зависимость, несколько итераций цикла могли бы выполняться параллельно, при условии, что каждая пара операторов в итерации поддерживается в заданном порядке.

Рассмотрим еще один, третий тип зависимостей по данным, который возникает в циклах, как показано в следующем примере:

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

Оператор S1 использует значение, которое присваивается оператором S2 в предыдущей итерации, так что имеет место зависимость между S2 и S1 между итерациями.

Несмотря на эту зависимость, этот цикл может быть сделан параллельным. Как и в более раннем цикле эта зависимость не циклическая: ни один из операторов не зависит сам от себя и, хотя S1 зависит от S2 в

предыдущей итерации, S2 не зависит от S1. Цикл является параллельным, если только отсутствует циклическая зависимость.

Хотя в вышеприведенном цикле отсутствуют циклические зависимости, чтобы выявить параллелизм и сделать его явным, цикл должен быть преобразован в другую структуру.

Здесь следует сделать два важных замечания:

1. Зависимость от S1 к S2 отсутствует. Если бы она была, то в зависимостях появился бы цикл и цикл не был бы параллельным. Вследствие отсутствия других зависимостей, перестановка двух операторов не будет влиять на выполнение оператора S2.
2. В первой итерации цикла оператор S1 зависит от значения B[1], вычисляемого перед началом цикла.

Эти два замечания позволяют заменить ранее рассмотренный цикл следующей последовательностью:

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

Теперь итерации цикла могут выполняться с перекрытием, при условии, что операторы в каждой итерации выполняются в заданном порядке. Имеется множество такого рода преобразований, которые реструктурируют цикл для выявления параллелизма.

Более подробно рассмотрим метод выявления параллелизма уровня команд. Зависимости по данным в откомпилированных программах представляют собой ограничения, которые оказывают влияние на то, какая степень параллелизма может быть использована. Вопрос заключается в том, чтобы подойти к этому пределу путем минимизации действительных конфликтов и связанных с ними приостановок конвейера.

Методы ориентированы на использование всего доступного параллелизма при поддержании истинных зависимостей по данным в коде программы.

Компилятор и аппаратура играют каждый свою роль: компилятор старается устранить или минимизировать зависимости, в то время как аппаратура старается предотвратить превращение зависимостей в приостановки конвейера.

8.4 Основы планирования загрузки конвейера и разворачивание циклов

Для поддержания максимальной загрузки конвейера должен использоваться параллелизм уровня команд, основанный на выявлении последовательностей несвязанных команд, которые могут выполняться в конвейере с совмещением.

Чтобы избежать приостановки конвейера зависимая команда должна быть отделена от исходной команды на расстояние в тактах, равное задержке конвейера для этой исходной команды.

Способность компилятора выполнять подобное планирование зависит:

- от степени параллелизма уровня команд, доступного в программе, и

• от задержки функциональных устройств в конвейере.

Будем рассматривать следующие задержки, определенные в таблице.

Таблица 8.4.1 Задержки выполнения команд

Команда, вырабатывающая результат	Команда, использующая результат	Задержка в тактах
Операция АЛУ с ПТ	Другая операция АЛУ с ПТ	3
Операция АЛУ с ПТ	Запись двойного слова	2
Загрузка двойного слова	Другая операция АЛУ с ПТ	1
Загрузка двойного слова	Запись двойного слова	0

Предположим, что условные переходы имеют задержку в **один такт,** так что команда, следующая за командой перехода не может быть определена в течение одного такта после команды условного перехода.

Предположим, что функциональные устройства полностью конвейеризованы или дублированы (столько раз, какова глубина конвейера), так что операция любого типа может выдаваться для выполнения в каждом такте и структурные конфликты отсутствуют.

Рассмотрим, каким образом компилятор может увеличить степень параллелизма уровня команд путем разворачивания циклов?

Для иллюстрации этих методов используем простой цикл, который добавляет скалярную величину к вектору в памяти:

```
For i:=1 to n do

A[i]:=A[i]+p
```

Очевидно, что это параллельный цикл, поскольку зависимость между итерациями цикла отсутствует. Предположим, что первоначально в регистре R1 находится адрес последнего элемента вектора (например, элемент с наибольшим адресом), а в регистре F2 - скалярная величина, которая должна добавляться к каждому элементу вектора. Программа для машины, не рассчитанная на использование конвейера, будет выглядеть примерно так:

```
Loop: LD F0,0(R1) ;   F0=элемент вектора
ADDD F4,F0,F2 ;       добавляет скаляр из F2
SD 0(R1),F4 ;         запись результата
SUBI R1,R1,#8 ;       пересчитать указатель ;8 байт (в двойном слове)
BNEZ R1, Loop ;       переход R1!=нулю
```

Для упрощения положим, что массив начинается с ячейки 0. Если бы он находился в любом другом месте, цикл потребовал бы наличия одной дополнительной целочисленной команды для выполнения сравнения с регистром R1.

Рассмотрим работу этого цикла при выполнении на простом конвейере с задержками, определенными выше.

Без планирования, работа цикла будет выглядеть следующим образом:

Такт выдачи

Loop: LD F0,0(R1)	**1**
приостановка	**2**
ADDD F4,F0,F2	**3**
приостановка	**4**
приостановка	**5**
SD 0(R1),F4	**6**
SUBI R1,R1,#8	**7**
BNEZ Rl,Loop	**8**
приостановка	**9**

Получили, что для выполнения цикла потребуется **9 тактов** на итерацию: одна приостановка для команды LD, две для команды ADDD, и одна для задержанного перехода.

Можно спланировать цикл иначе, чтобы получить меньшее время для выполнения:

Loop: LD F0,0(R1)	**1**
Приостановка	**2**
ADDD F4,F0,F2	**3**
SUBI R1,R1,#8	**4**
BNEZ Rl,Loop ;	**задержанный переход 5**
SD 8(R1),F4 ;	**команда изменяется, когда 6**

;меняется местами с командой SUBI

Время выполнения уменьшилось с **9 до 6 тактов!**

Заметим, что для планирования задержанного перехода компилятор должен определить, что он может поменять местами команды SUB 1 и SD путем изменения адреса в команде записи SD: Адрес был равен 0(R1), а теперь равен 8(R1). Это не тривиальная задача, поскольку большинство компиляторов будут видеть, что команда SD зависит от SUB1, и откажутся от такой перестановки мест. Более изощренный компилятор смог бы рассчитать отношения и выполнить перестановку. Цепочка зависимостей от команды LD к команде ADDD и далее к команде SD определяет количество тактов, необходимое для данного цикла.

В вышеприведенном примере мы завершаем одну итерацию цикла и выполняем запись одного элемента вектора каждые 6 тактов, но действительная работа по обработке элемента вектора отнимает только 3 из этих 6 тактов (загрузка, сложение и запись). Оставшиеся 3 такта составляют накладные расходы на выполнение цикла (команды SUBI, BNEZ и приостановка). Чтобы устранить эти три такта, нам нужно иметь больше операций в цикле относительно числа команд, связанных с накладными расходами.

Одним из наиболее простых методов увеличения числа команд по отношению к команде условного перехода и команд, связанных с накладными расходами, является разворачивание цикла. Такое разворачивание выполняется путем многократной репликации (повторения) тела цикла и коррекции соответствующего кода конца цикла.

Разворачивание циклов может также использоваться для улучшения планирования. В этом случае, мы можем устранить приостановку, связанную с задержкой команды загрузки путем создания дополнительных независимых комащ в теле цикла. Затем компилятор может планировать эти команды для помещения в слот задержки команды загрузки. Если при разворачивании цикла мы просто реплицируем команды, то результирующие зависимости по именам могут помешать эффективно спланировать цикл. Таким образом, для разных итераций хотелось бы использовать различные регистры, что увеличивает требуемое число регистров.

Развернем рассмотренный выше цикл так, сделав *четыре* копии тела цикла, предполагая, что R1 первоначально кратен 4. Устраним при этом любые очевидные излишние вычисления и не будем пользоваться повторно никакими регистрами.

Ниже приведен результат, полученный **путем слияния команд SUB1 и выбрасывания ненужных операций BNEZ, которые дублируются при разворачивании цикла.**

Loop: LD F0,0(R1) (2 такта)
ADDD F4,F0,F2 (3 такта)
SD 0(R1),F4 выбрасывается SUB1 и BNEZ
LD F6,-8(R1) (2 такта)
ADDD F8,F6,F2 (3 такта)
SD -8(R1),F8 выбрасывается SUB1 и BNEZ
LDF10,-16(R1) (2 такта)
ADDD F12,F10,F2 (3 такта)
SD -16(R1),F12 выбрасывается SUB1 и BNEZ
LDF14,-24(R1) (2 такта)
ADDD F16,F14,F2 (3 такта)
SD -24(R1),F16
SUB1 R1,R1,#32
BNEZ Rl, Loop (2 такта)

Мы *ликвидировали три условных перехода и три операции декрементирования R1*.

Адреса команд загрузки и записи были скорректированы так, чтобы позволил слить команды SUB1 в одну команду по регистру R1. При отсутствии планирования, за каждой командой здесь следует зависимая команда и это будет приводить к приостановкам конвейера.

Этот цикл будет выполняться за 27 тактов:(на каждую команду LD потребуется 2 такта, на каждую команду ADDD - 3, на условный переход - 2 и на все другие команды 1 такт) или по 6.8 такта на каждый из четырех элементов. Полученная развернутая версия в такой редакции медленнее, чем

оптимизированная версия исходного цикла, однако, после оптимизации самого развернутого цикла ситуация изменится.

Обычно разворачивание циклов выполняется на более ранних стадиях процесса компиляции, так что избыточные вычисления могут быть выявлены и устранены оптимизатором.

Если аналогично вышеприведенному примеру рассмотреть пример цикла для суммирования двух векторов, то накладные расходы на организацию цикла составят около 0.5 времени выполнения. Кроме того команды тела цикла зависят по управлению от команд управления циклом. Одним из наиболее простых методов увеличения числа команд по отношению к командам, связанным с накладными расходами, и является разворачивание цикла (многократное повторение операторов и соответствующая коррекция завершения).

В реальных программах мы обычно не знаем верхней границы цикла. Предположим, что она равна п и мы разворачиваем цикл так, чтобы иметь **к** копий тела цикла. Тогда, вместо единственного развернутого цикла мы генерируем пару циклов. Первый из них выполняется (n mod k) раз и имеет тело первоначального цикла. Развернутая версия цикла окружается внешним циклом, который выполняется (n div k) раз.

В вышеприведенном примере разворачивание цикла увеличивает его производительность путем устранения команд, связанных с накладными расходами цикла, хотя при этом заметно увеличивается размер программного кода.

Насколько увеличится производительность, если цикл оптимизировать?

Ниже представлен развернутый цикл из предыдущего примера после оптимизации.

```
Loop: LD F0,0(R1)
LD F6,-8(R1)
LDF10,-16(R1)
LDF14,-24(R1)
ADDD F4,F0,F2
ADDD F8,F6,F2
ADDDF12,F10,F2
ADDDF16,F14,F2
SD 0(R1),F4
SD -8(R1),F8
SD -16(R1),F12
SUB1 R1,R1,#32
BNEZ Rl, Loop
SD 8(R1),F16 ; 8 - 32 = -24
```

Время выполнения развернутого цикла снизилось до 14 тактов или до 3.5 тактов на элемент, по сравнению с 6.8 тактов на элемент до оптимизации, и по сравнению с 6 тактами при оптимизации без разворачивания цикла.

Выигрыш от оптимизации развернутого цикла даже больше, чем от оптимизации первоначального цикла! Это произошло потому, что

разворачивание цикла выявило больше вычислений, которые могут быть оптимизированы для минимизации приостановок конвейера (приведенный выше программный код выполняется без приостановок). При подобной оптимизации цикла необходимо осознавать, что команды загрузки и записи являются независимыми и могут чередоваться. Анализ зависимостей по данным позволяет нам определить, являются ли команды загрузки и записи независимыми.

Разворачивание циклов представляет собой простой, но полезный метод увеличения размера линейного кодового фрагмента, который может эффективно оптимизироваться. Это преобразование полезно на множестве машин от простых конвейеров, подобных рассмотренному ранее, до суперскалярных конвейеров, которые обеспечивают выдачу для выполнения более одной команды в такте.

Методы, которые используются аппаратными средствами для динамического планирования загрузки конвейера и сокращают приостановки из-за конфликтов типа RAW(Read After Write) (аналогичны рассмотренным выше методам компиляции).

8.5 Устранение зависимостей по данным и механизмы динамического планирования

Главным ограничением методов конвейерной обработки - является выдача для выполнения команд строго в порядке, предписанном программой:

если выполнение какой-либо команды в конвейере приостанавливалось, следующие за ней команды также приостанавливались.

Таким образом, при наличии зависимости между двумя близко расположенными в конвейере командами возникала приостановка обработки многих команд. Но если имеется несколько функциональных устройств, многие из них могут оказаться незагруженными. Если команда] зависит от длинной команды i, выполняющейся в конвейере, то все команды, следующие за командой j должны приостановиться до тех пор, пока команда i не завершится и не начнет выполняться команда]. Например, рассмотрим следующую последовательность команд:

DIVD F0,F2,F4
ADDD F10,F0,F8
SUBD F8,F8,F14

Команда SUBD не может выполняться из-за того, что зависимость между командами DIVD и ADDD привела к приостановке конвейера. Однако команда SUBD не имеет никаких зависимостей от команд в конвейере. Это ограничение производительности, которое может быть устранено *снятием требования о выполнении команд в строгом порядке.*

В рассмотренном конвейере структурные конфликты и конфликты по данным проверялись во время стадии декодирования команды (ID). Если команда могла нормально выполняться, она выдавалась с этой ступени конвейера в следующие. Чтобы позволить начать выполнение команды

SUBD из предыдущего примера, необходимо разделить процесс выдачи на две части: *проверку наличия структурных конфликтов* и *ожидание отсутствия конфликта по данным*.

Когда мы выдаем команду для выполнения, мы можем осуществлять проверку наличия структурных конфликтов. Таким образом, мы все еще используем упорядоченную выдачу команд.

Однако мы хотим начать выполнение команды, как только станут доступными ее операнды. Таким образом, конвейер будет осуществлять неупорядоченное выполнение команд, которое означает и неупорядоченное завершение команд.

Неупорядоченное завершение команд создает основные трудности при обработке исключительных ситуаций.

В машинах с динамическим планированием потока команд прерывания будут неточными, поскольку команды могут завершиться до того, как выполнение более ранней выданной команды вызовет исключительную ситуацию. Таким образом, очень трудно повторить запуск после прерывания.

Чтобы *реализовать неупорядоченное выполнение команд, мы расщепляем ступень ID на две ступени:*

1. Выдача - декодирование команд, проверка структурных конфликтов.

2. Чтение операндов - ожидание отсутствия конфликтов по данным и последующее чтение операндов.

Затем, следует ступень EX. Поскольку выполнение команд ПТ может потребовать нескольких тактов в зависимости от типа операции, мы должны знать, когда команда начинает выполняться и когда заканчивается. Это позволяет нескольким командам выполняться в один и тот же момент времени.

В дополнение к этим изменениям структуры конвейера можно изменить

• структуру функциональных устройств, варьируя количество устройств, задержку операций

• степень конвейеризации функциональных устройств так, чтобы лучше использовать эти методы конвейеризации.

Динамическая оптимизация с централизованной схемой обнаружения конфликтов реализуется аппаратно.