Exploiting ILP with Software Approaches

陈文智
chenwz@zju.edu.cn
浙江大学计算机学院
2014年12月

4.1 Basic Compiler Techniques for Exposing ILP

4.1.1 流水线调度

- 流水线调度：若j指令要用到I指令的结果(RAW相关)，流水线调度是指把（i和j)两条指令分隔开来，两者之间应间隔的时钟周期数等于源指令产生结果所需的延时(latency)时钟周期数。
- 两种流水线调度方法
  - Static pipeline scheduling by compiler
  - Dynamic pipeline scheduling by hardware

本章采用如下假设

- 无结构竞争(即有足够硬件可供使用),每一时钟周期可发射一条指令
- 采用MIPS标准的整数操作流水线结构(即由IF,ID,EX,MEM,WB五拍组成)
- 转移指令(Branch)后有一个时钟周期延时
- 浮点操作的延时时钟周期数参见表

浮点操作延迟时间

前操作指令	后继相关指令	等待时钟周期数
FP ALU 操作	FP ALU 操作	3
FP ALU 操作	Store(双字)	2
Load(双字)	FP ALU 操作	1
Load(双字)	Store(双字)	0

4.1.2 Loop Unrolling

- 通过编译循环展开和指令序列调度来提高流水线性能

```
For (i=1; i<=1000; i++)
        x[i] = x[i] +s;
```

其中: **x[i] ----array element;**

s ---- scalar.

转换为MIPS汇编语言代码如下:

```
Loop: L.D   F0, 0(R1)
      ADD.D  F4, F0, F2
      S.D    0(R1),  F4
      DADDUI   R1, R1, #-8
      BNEZ   R1, R2,Loop
```

R1: 数组元素的地址,初始化为最高地址

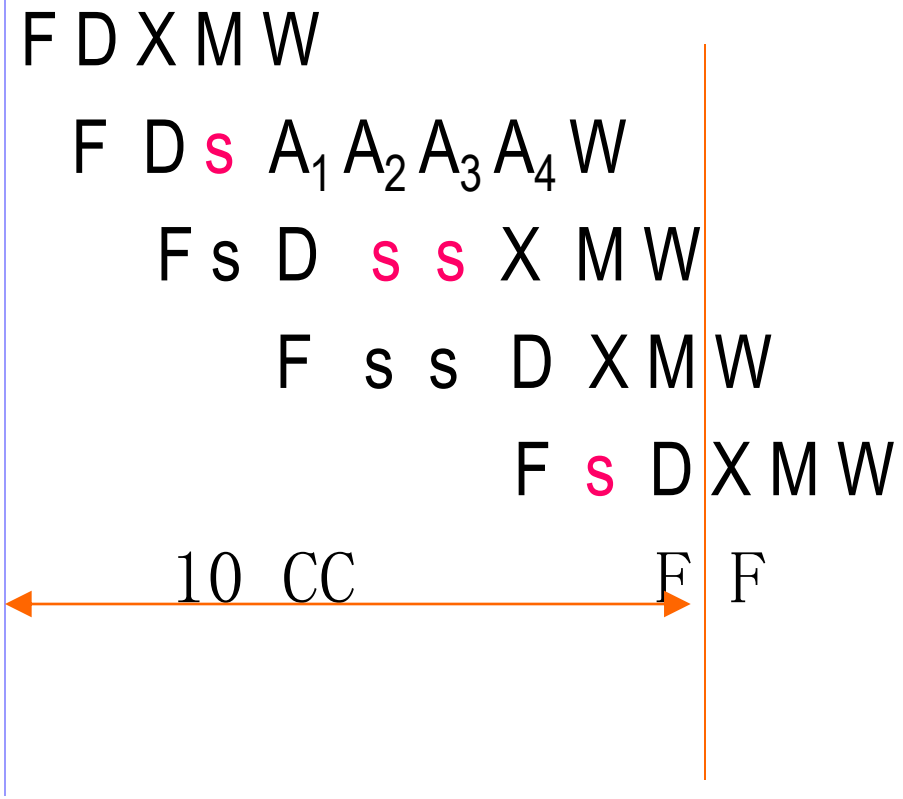
F2: 标量值s

双字地址间隔为**8**字节

R2: 预先计算，**8(R2)**最后一个元素。

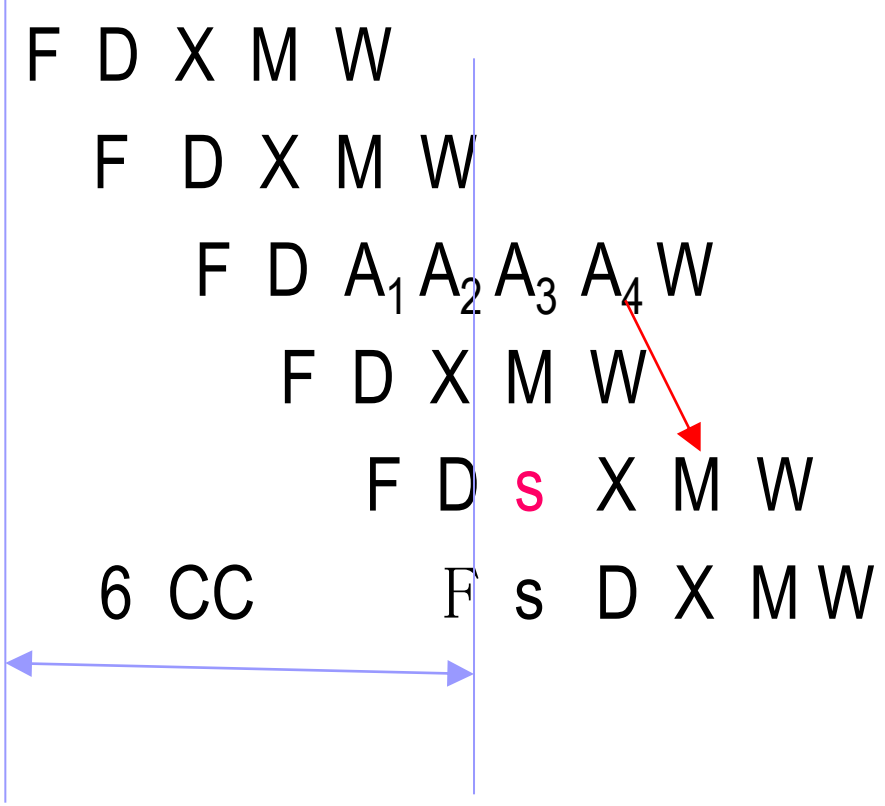
一、未调度时loop一次迭代所需时钟周期数

```
Loop: LD   F0, 0(R1)
      ADDD F4, F0, F2
      SD   0(R1),  F4
      SUBI R1, R1, 8
      BNEZ R1, Loop
```



二 对loop代码调度后,一次迭代所需时钟周期数

```
Loop: LD   F0, 0(R1)
      SUBI R1, R1, 8
      ADDD F4, F0, F2
      BNEZ R1, Loop
      SD   8(R1), F4
```



前页说明:

- 为了颠倒SUBI和SD顺序,SD的地址发生了改变! 即恢复到原地址
- 一次迭代由10个时钟周期减少到6个时钟周期
- 实际上算一个数组元素仅需3个时钟周期(Load,add和store);另外3个时钟周期(SUBI,BNEZ和一个stall)是loop的开销

三、loop unrolling (消除loop overhead)

```
Loop: LD F0, 0(R1)
      stall
      ADDD F4, F0, F2
      stall, stall
      SD     0(R1), F4
      LD     F6, -8(R1)
      stall
      ADDD F8, F6, F2
      stall, stall
      SD     -8(R1), F8
      LD     F10, -16(R1)
      stall
      ADDD F12, F10, F2
      stall, stall
      SD     -16(R1), F12
      LD     F14, -24(R1)
      stall
      ADDD F16, F14, F2
      stall, stall
      SD     -24(R1), F16
      SUBI   R1, R1, #32
      stall
      BNEZ   R1, loop
      stall
```

前页说明

- 假设循环总数是4的倍数，我们将Loop展开4次
- SUBI指令中R1要减32
- 注意loop展开后,每一次迭代采用不同寄存器,如用F0, F6, F10, F14表示LD的目的寄存器,分别表示不同变量
- 展开后loop需28个时钟周期,即每次迭代平均需28/4=7个时钟周期,仅通过展开,消除loop overhead, 就可缩短每次迭代的时钟周期数, 这里没有做任何调度.

四、对unrolling loop进行调度,达到进一步缩短每次迭代的时钟周期数

```
Loop: LD F0, 0(R1)
      LD F6, -8(R1)
      LD F10, -16(R1)
      LD F14, -24(R1)
      ADDD F4, F0, F2
      ADDD F8, F6, F2
      ADDD F12, F10, F2
      ADDD F16, F14, F2
      SD   0(R1), F4
      SD   -8(R1), F8
      SUBI R1, R1, #32
      SD   -16(R1), F12
      BNEZ R1, Loop
      SD   8(R1), F16
```

(因R1已减32, 所以加8)

前页说明:

- 展开调度后的Loop共需14个时钟周期,则每次迭代平均只需14/4=3.5个时钟周期
- 调度展开的循环对提高性能的作用大于单纯的调度

例子说明的问题:

- 通过例子,我们看到研究开发ILP对提高处理器功能单元性能(即流水线性能)的巨大作用;
- 流水线思想早在60年代就开始应用于处理器,但只有在80年代和90年代,在深入研究ILP之后提出一系列先进流水线技术,才成为使微处理器性能突飞猛进的关键技术;
- 上述例子所采用的一些方法对我们人类来讲都十分直观和简单,但要使硬件和软件(编译器)来完成上述过程,必须总结出一套形式化的,方法学上的条例来确定何时以及如何来改变指令的执行顺序。

Summary： 关于循环展开和调度方法在执行过程中，我们做出了以下决策和代码变换。

(1) 确信把SD移到SUBI和BNEZ之后是合法的，并求出SD的位移量；

(2) 确信循环体的每次迭代是相互独立的（除维持循环的代码外），以及循环体展开有利于性能提高；

(3) 为了避免因采用同一寄存器而造成不必要的限制，可以采用不同寄存器表示不同变量；

- Renaming with software

(4) 消除额外测试和转移指令，调整维持循环的代码；

(5) 只有确信不同迭代中的Loads和stores是互相独立的之后，Loads和Stores才能在展开后的循环体中互换位置。为此必须分析存储器的地址，并确信Loads和stores访问的并非同一地址。这就是所谓的memory disambiguation。

(6) 在调度指令执行顺序时，必须确保相关性不变，才能使调度后的代码的结果与源代码的相同。

4.1.3 Using Loop Unrolling and Pipeline Scheduling with Static Multiple Issue

- **To schedule this loop without any delays, we will need to unroll the loop to make five copies of the body. After unrolling, the loop will contain five each of `L.D`, `ADD.D`, and `S.D`; one `DADDUI`; and one `BNE`. The unrolled and scheduled code is shown below.**

	Integer instruction		FP instruction	Clock cycle
Loop:	L.D	F0,0(R1)		1
	L.D	F6,-8(R1)		2
	L.D	F10,-16(R1)	ADD.D F4,F0,F2	3
	L.D	F14,-24(R1)	ADD.D F8,F6,F2	4
	L.D	F18,-32(R1)	ADD.D F12,F10,F2	5
	S.D	F4,0(R1)	ADD.D F16,F14,F2	6
	S.D	F8,-8(R1)	ADD.D F20,F18,F2	7
	S.D	F12,-16(R1)		8
	DADDUI	R1,R1,#-40		9
	S.D	F16,16(R1)		10
	BNE	R1,R2,Loop		11
	S.D	F20,8(R1)		12

- 超标量结构执行展开调度后的Loop共需12个时钟周期,则每次迭代平均只需12/5=2.4个时钟周期
- 多发射在编译优化下对提高性能的作用非常大（6，3.5）
  - Speedup=6/2.4=2.5
  - Speedup=3.5/2.4=1.458
- 从10--> 6 --> 3.5 --------> 2.4 ----------> 1.29

(调度) (展开+调度) (多发射+展开+调度）

4. 2 Static Multiple Issue: the VLIW Approach

4.2.1 VLIW处理器的特点

1、一次发射一条长指令，其中包含多个操作，而不是像超标量处理器那样一次发射多条指令。这样做可以减轻指令发射逻辑电路的带宽，因为超标量处理器中为了发射多条指令的需要，必须将指令发射逻辑电路流水化，并提高其带宽，使其硬件复杂化，同时增加了成本。

VLIW处理器的特点（2）

2、长指令的组装由Compiler完成，而不需要像超标量处理器那样由动态调度硬件完成，从而进一步减轻硬件负担，当然也丧失了动态调度的优点。

所以：VLIW与Superscalar相比较，硬件相对简单、廉价。

VLIW处理器的特点（3）

3、为了使所有功能单元充分发挥作用，必须要开发更多的指令并行性，即有足够多能并行执行的指令去填充VLIW。这里要采用全局调度技术（Globle scheduling technique)。

- 全局调度技术：即跨越条件转移指令的调度技术，包含：循环展开，跨越基本块的调度，路径调度（Trace scheduling)等技术。

4.2.2 VLIW处理器实例

- 设处理器含5个功能单元：
  - 1个整数操作和转移指令部件
  - 2个FP操作部件
  - 2个访存操作部件
- 实现数组加法的实例：

  X[i] = X[i] + s ;

  Unroll as many times as necessary to eliminate any stalls. Ignore the branch-delay slot.
- **The loop has been unrolled to make seven copies of the body, which eliminates all stalls (i.e., completely empty issue cycles), and runs in 9 cycles. This code yields a running rate of seven results in 9 cycles, or 1.29 cycles per result.**

VLIW展开循环、封装指令的结果

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D -8(R1),F8	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D -24(R1),F16			
S.D F20,-32(R1)	S.D -40(R1),F24			DADDUI R1,R1,#-56
S.D F28,8(R1)				BNE R1,R2,Loop

- This code takes nine cycles assuming no branch delay; normally the branch delay would also need to be scheduled. The issue rate is 23 operations in nine clock cycles, or 2.5 operations per cycle. The efficiency, the percentage of available slots that contained an operation, is about 60%.
- To achieve this issue rate requires a larger number of registers than MIPS would normally use in this loop. The VLIW code sequence above requires at least eight FP registers, while the same code sequence for the base MIPS processor can use as few as two FP registers or as many as five when unrolled and scheduled. In the superscalar, six registers were needed.

4.2.3 多发射处理器的局限性

既然可以在一个时钟周期内发射5条指令，那么为什么不同时发射50条指令呢？

多发射方法的困难由哪些？

- 存在三方面困难：
  - 程序中固有ILP有限；
  - 多发射处理器硬件复杂性高，成本高；
  - Superscalar和VLIW实现的专有困难。

多发射方法的困难（1）

- 程序固有ILP有限是多发射处理器的本质困难
  - 我们需要的可并行的指令数并非等于功能单元数就能满足。
  - 一般，我们需要的可并行（即独立的）指令数大致等于功能单元数乘以平均流水级数。这是因为这些功能单元中，如存储器访问，转移指令，FP 操作都是流水化的，或有一定延时。

多发射方法的困难（2）

- 硬件复杂，成本高

  对多发射处理器来讲，在多指令发射和执行方面，既要增加发射和执行的硬件数量，又要提高它们的带宽（速度，即性能）。

多发射方法的困难（3）

- Superscalar的特殊困难
  - 发射逻辑复杂且高速
  - 动态调度硬件更复杂化

多发射方法的困难（4）

- VLIW的特殊困难
  - 造成代码（code)量增大。因为VLIW指令中有很多域是未填满的，造成浪费码长；
  - 一条长指令中任一个功能单元的锁操作（即停顿）造成所有功能单元停顿。因为所有功能单元都是同步工作的。
  - 对VLIW系列处理器，二进制代码兼容困难。系列机中发射指令数目和功能单元延时长短不一，造成系列机中二进制代码不兼容。

4.3 Advanced Compiler Support for Exposing and Exploiting ILP

- **Loop unrolling ——将loop展开形成长代码序列的方法。**
- **软件流水技术——software pipelining，又称象征性循环展开技术（symbolic loop unrolling）。**
- **路径调度技术——trace scheduling**

4.3.1 Software Pipelining: Symbolic Loop Unrolling

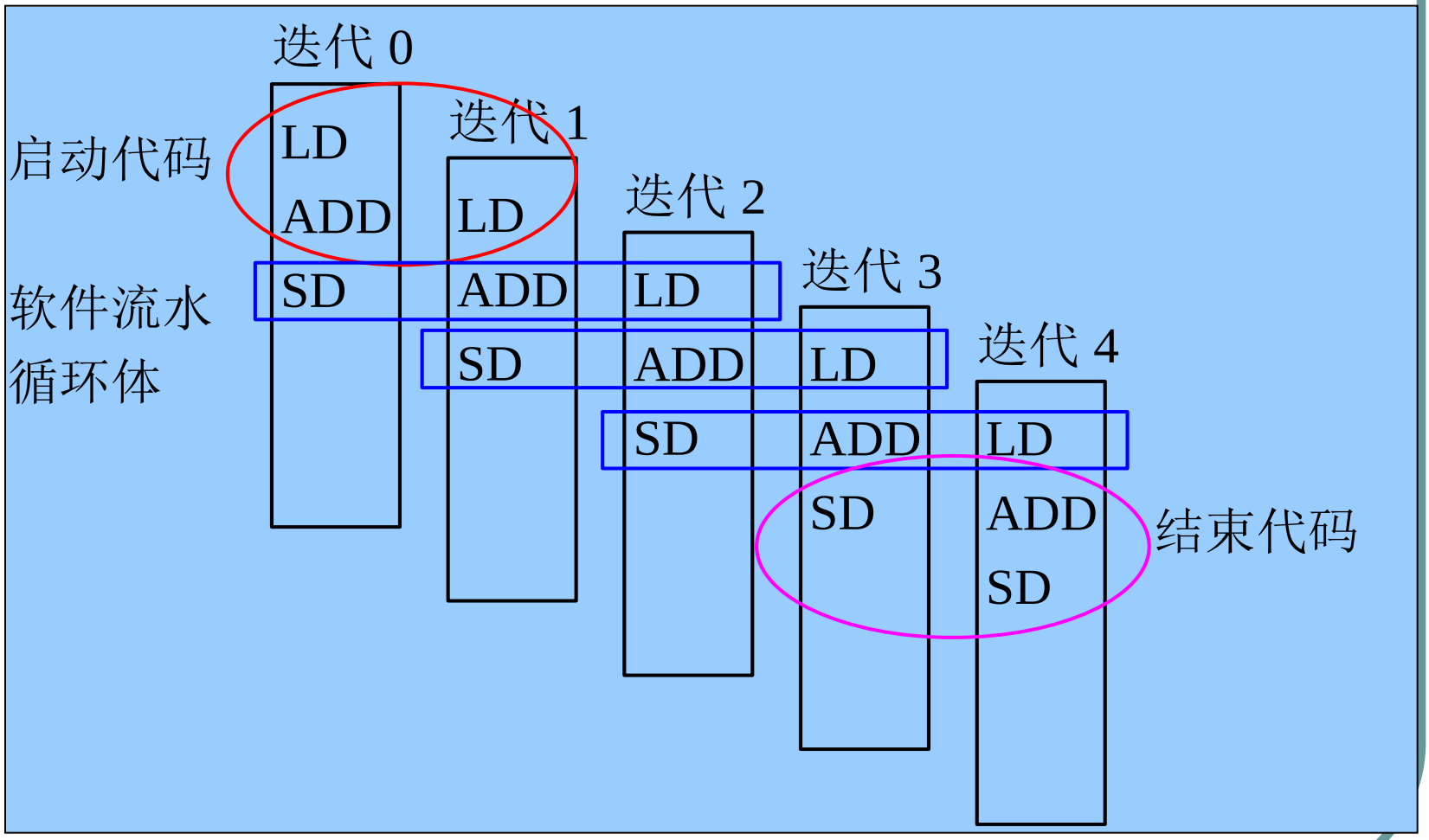
软件流水线技术是一种重组循环体的技术，在软件流水循环（***software* pipeline loop**）的每一次迭代体（即新的重组后的循环）是由从原循环的不同迭代中选出的指令组成的，可以达到消除相关性引起竞争的目的。

软件流水循环体是如何组成的？

- 这里新的软件流水**loop**是通过**symbolic loop unrolling,**即象征性地循环展开方法来获得的.

```
loop:   L.D      F0, 0(R1)
        ADD.D    F4, F0, F2
        S.D      0(R1), F4
        DADDUI   R1, R1,#-8
        BNEZ     R1, loop
```

新软件流水loop组成

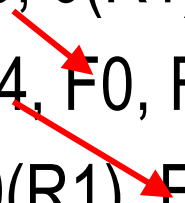


新循环结构

- 这里并非按传统方法展开循环体，而是象征性地从原loop每一次迭代中选取一条指令
  - 原来loop由5条指令组成，其中两条为loop开销。
  - 新软件流水循环体也由5条指令组成
    - 包含两条循环开销指令。
    - 其中三条指令是象征性地从每一迭代中选取。
  - 有一段起始代码段（startup code）
  - 有一段结束代码段（finish-up code）。

原循环相关性分析

```
loop: L.D      F0, 0(R1)
      ADD.D F4, F0, F2
      S.D      0(R1), F4
      DADDUI   R1, R1,#-8
      BNEZ  R1, loop
```



存在RAW相关性(F0)

存在RAW相关性(F4)

新软件流水循环体

- 象征性展开loop：

迭代i:　LD
M[i]　ADDD
　　SD
迭代i+1: LD
M[i-1]　ADDD
　　SD
迭代i+2:LD
M[i-2]　ADDD
　　SD

地址相差16

```
L.D     F0, 0(R1)
ADD.D F4, F0, F2
DADDUI  R1, R1,#-8
L.D     F0, 0(R1)
DADDUI  R1, R1,#-8
```

- 从每一次迭代中选一条指令组成新的循环体

```
loop: S.D     F4 ,16(R1);存到M[i ]
      ADD.D   F4, F0, F2;M[i-1]
      LD      F0, 0(R1) ;取M[ i-2 ]
      DADDUI  R1, R1, #-8
      BNEZ    R1, loop
```

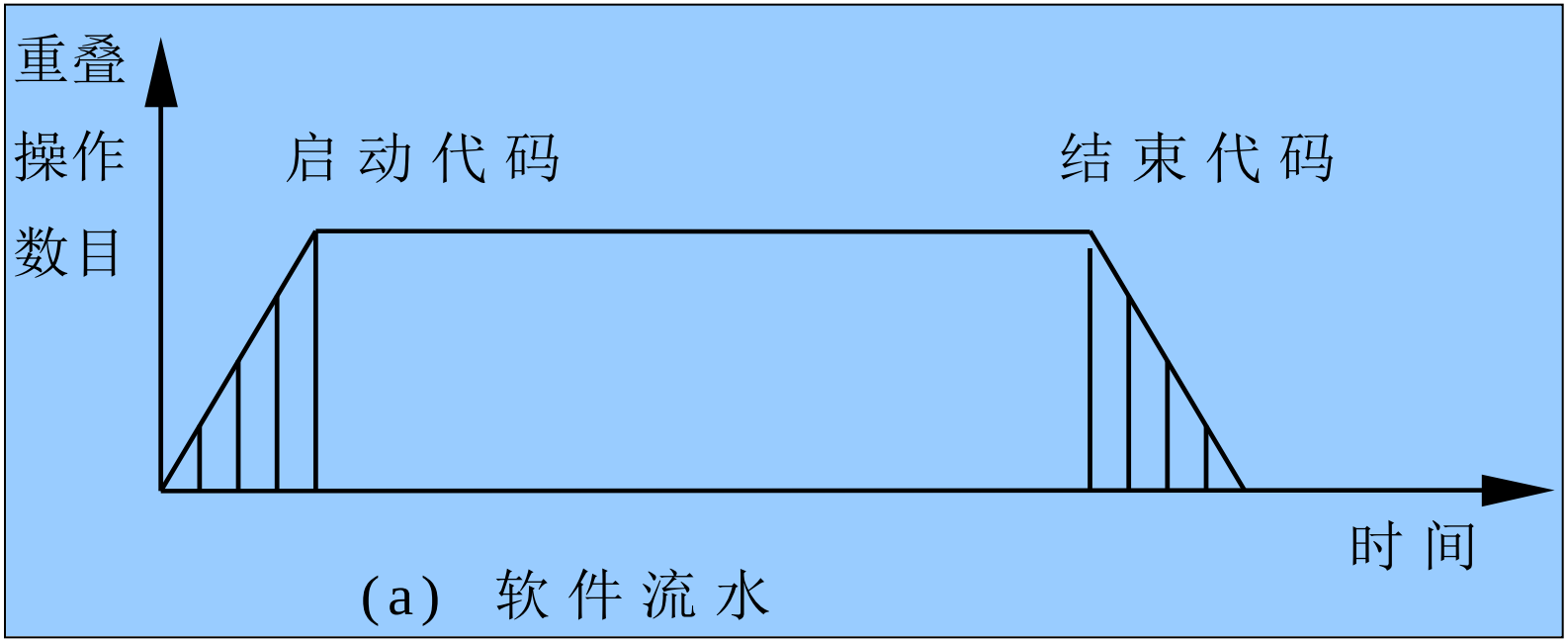
```
S.D     F4, 16(R1)
ADD.D  F4,F0,F2
S.D     F4,8(R1)
```

小结：

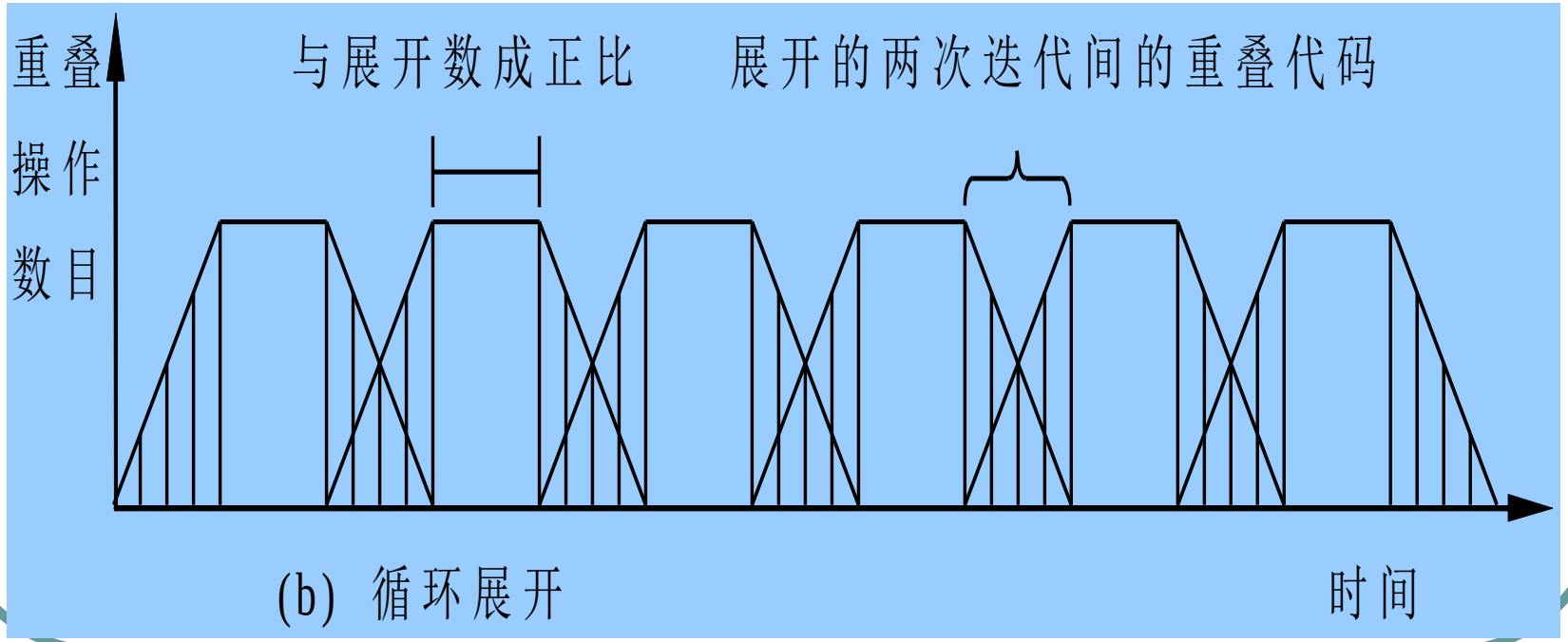
- 软件流水线技术的实质
  - 仅仅是一种调度方法，即象征性循环展开方法
- 软件流水技术与直接循环展开相比具有以下优点
  - 占用较小的代码空间，因为不必象直接循环展开那样需形成一条较长的代码序列；
  - 减少了loop头尾部分的开销。

例：若有一loop含100次迭代，

- 采用软件流水线技术
  - 减少的是循环执行时间



- 采用直接循环展开
  - 减少的是循环开销（分支和计数器更新的代码）。
  - 由于不可能将100次迭代全部直接展开，（因为占用太多代码空间，寄存器），通常，每次展开几个迭代，这样能流水化（即重叠执行）的指令数就远远少于软件流水技术。



4.3.2 Global Code Scheduling trace scheduling --路径调度

- 按关键路径进行调度(Using critical path scheduling)
- 适用范围:
  - 路径调度是为开发条件转移之间的ILP而提出的。
  - 循环展开是为开发Loop之间的ILP而提出的。

*路径调度技术的提出：

- 迄今我们研究了程序基本块内，loop间的ILP开发，然而多发射处理器的引入，要求开发更多的ILP来发挥其性能。因此，进一步研究包含条件转移的程序基本块之间的ILP----路径调度技术。

*路径调度的基本思想

- 路径调度技术应用转移预测技术和投机执行思想，预测代码执行方向（即预测条件转移指令的转移方向），也就是预测代码的执行路径。选择转移概率大的执行路径，将这条路径上的基本块合并在一起，扩大基本块的规模。然后投机执行（即提前执行）。这样做的结果是突破了转移指令造成的线性代码块规模的限制，从而增加了进一步开发出更多ILP的可能性。

*路径调度方法：

有三个不同过程组成：

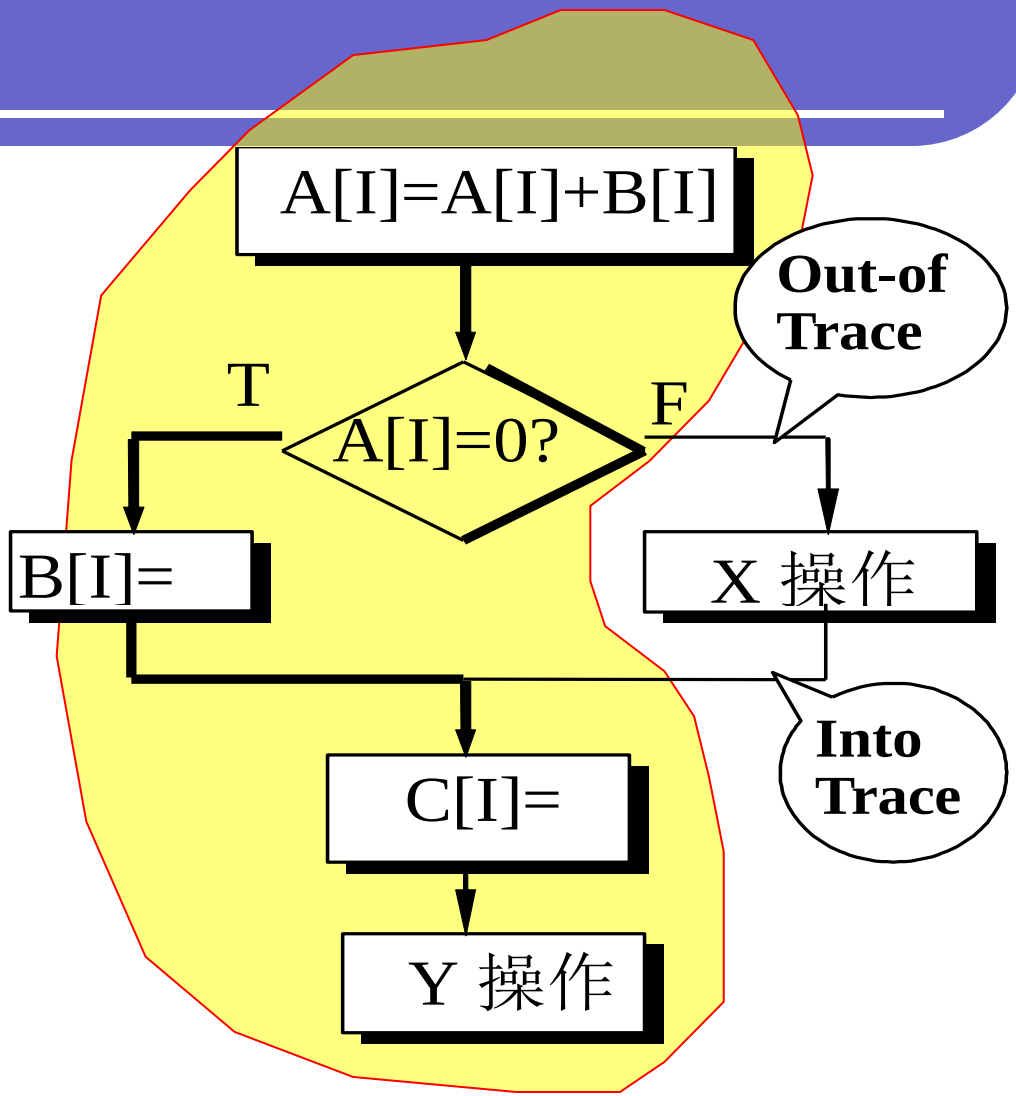
- Trace selection（路径选择）
- Trace compaction（路径压缩）
- Predict miss compensate(路径补偿)

*路径选择

遇到条件转移指令时，首先预测转移的可能方向，将转移概率大的方向选作代码的执行路径，将转移指令前的代码基本块和转移指令后的路径上的基本块合并成一个大的基本块。若新形成的大基本块中含有**loop**，则展开该**loop**，最终形成一条长的代码序列，供进一步开发**ILP**。

路径选择图示：

```
A[I] = A[I]+B[I];
IF A[I] = 0
      B[I] =  …  ;
ELSE{
      ……
      X 操作;
      ……
}
C[I] =  …  ;
   Y 操作;
    ……
```



- 若True概率大，则将左边分支选作Trace,统一开发其ILP

*路径压缩

一旦路径选好后，可以将转移指令后的代码提前到转移指令前执行（即投机执行）。根据所采用的多发射方法，将路径内的代码组织成超长指令字，或超标量指令，提高ILP。经过压缩，使代码长度缩短，这就是“compaction”的含义。由此可知，路径压缩是一种整体调度（Global schedule）。

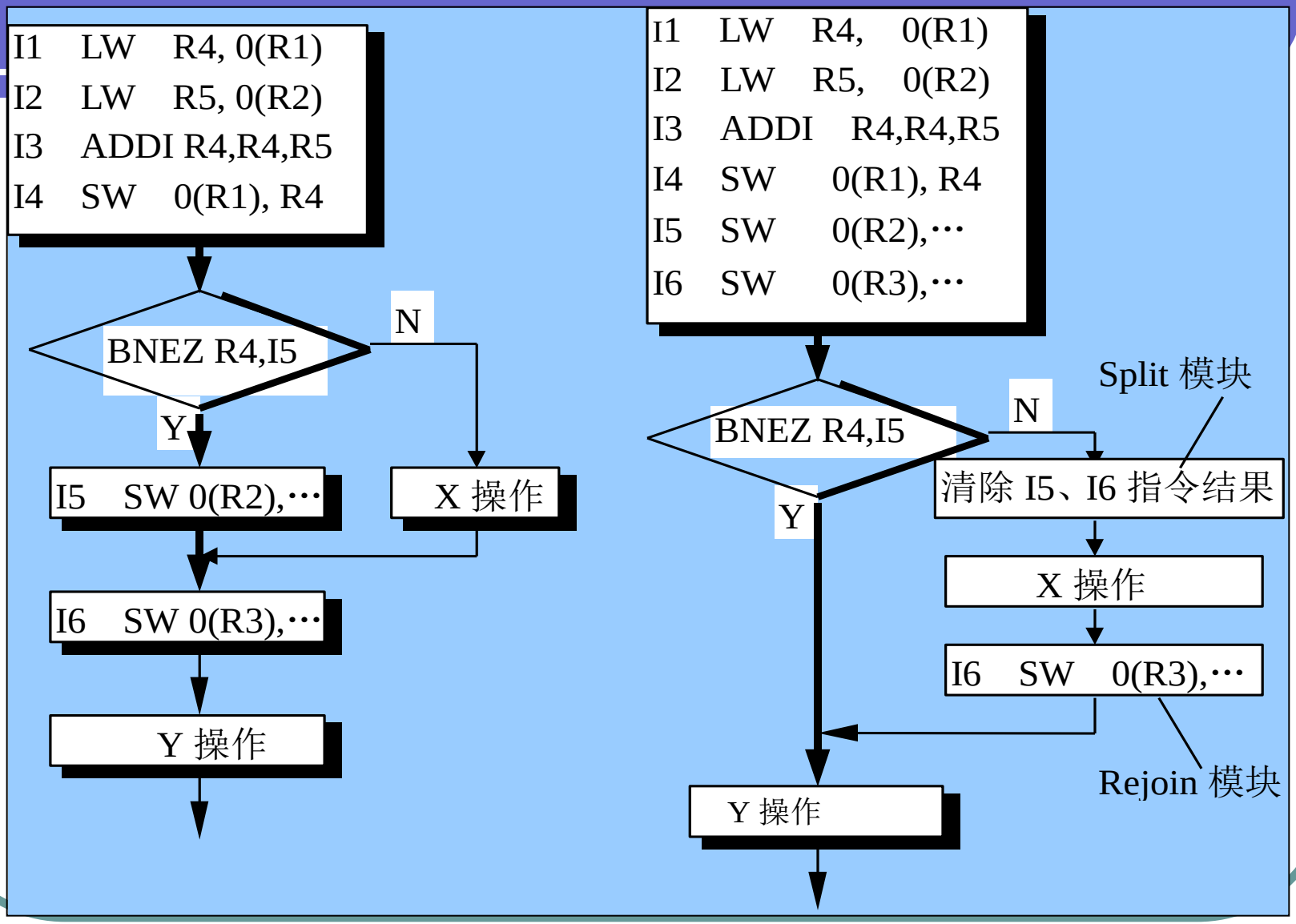
路径压缩要注意两件事

- 保证data dependence
  - 可采用前面已介绍过的循环展开相关性分析和静动态调度方法来解决；
- 保证控制相关性
  - 若预测是对的，则所选择的路径不会破坏**control dependence**，若预测是错的，则破坏了控制相关性，则必须给予补偿和调整执行路径。

路径补偿

- 包含两个工作，分别对应branch指令转出路径口（branch out-of trace）和branch指令转入路径口（branch into trace）处的两项工作：
  - 在Branch out-of trace处，要清除投机执行指令（B[i]和C[i]）的结果；
  - 在Branch into trace处要重新执行C[j]赋值语句。

例


I1 LW R4, 0(R1)
I2 LW R5, 0(R2)
I3 ADDI R4,R4,R5
I4 SW 0(R1), R4
BNEZ R4,I5
N
Y
I5 SW 0(R2),…
X 操作
I6 SW 0(R3),…
Y 操作
I1 LW R4, 0(R1)
I2 LW R5, 0(R2)
I3 ADDI R4,R4,R5
I4 SW 0(R1), R4
I5 SW 0(R2),…
I6 SW 0(R3),…
BNEZ R4,I5
N
Y
Split 模块
清除 I5、I6 指令结果
X 操作
I6 SW 0(R3),…
Rejoin 模块
Y 操作