计算机体系结构

周学海
xhzhou@ustc.edu.cn
0551-63606864
中国科学技术大学

Review：Chapter04 存储系统

- **重点关注：存储层次、Cache性能评估和优化、主存组织、虚拟存储（TLB）**
- **Cache基本原理 4Q**
- **Cache性能分析**

CPU time = (CPU execution clock cycles +
Memory stall clock cycles) x clock cycle time

Memory stall clock cycles =
(Reads x Read miss rate x Read miss penalty +
Writes x Write miss rate x Write miss penalty)

Memory stall clock cycles =
Memory accesses x Miss rate x Miss penalty

Different measure: AMAT
Average Memory Access time (AMAT) =
Hit Time + (Miss Rate x Miss Penalty)

Cache优化方法：基本（6种）+高级（10种）

Review：10种高级优化方法

Technique	Hit time	Band-width	Miss penalty	Miss rate	Power consumption	Hardware cost/ complexity	Comment
Small and simple caches	+			–	+	0	Trivial; widely used
Way-predicting caches	+				+	1	Used in Pentium 4
Pipelined cache access	–	+				1	Widely used
Nonblocking caches		+	+			3	Widely used
Banked caches		+			+	1	Used in L2 of both i7 and Cortex-A8
Critical word first and early restart			+			2	Widely used
Merging write buffer			+			1	Widely used with write through
Compiler techniques to reduce cache misses				+		0	Software is a challenge, but many compilers handle common linear algebra calculations
Hardware prefetching of instructions and data			+	+	–	2 instr., 3 data	Most provide prefetch instructions; modern high-end processors also automatically prefetch in hardware.
Compiler-controlled prefetching			+	+		3	Needs nonblocking cache; possible instruction overhead; in many CPUs

Figure 2.11 Summary of 10 advanced cache optimizations showing impact on cache performance, power consumption, and complexity. Although generally a technique helps only one factor, prefetching can reduce misses if done sufficiently early; if not, it can reduce miss penalty. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.

Review：DRAM芯片及DRAM模块

- **存储器组织**
  - 存储系统结构

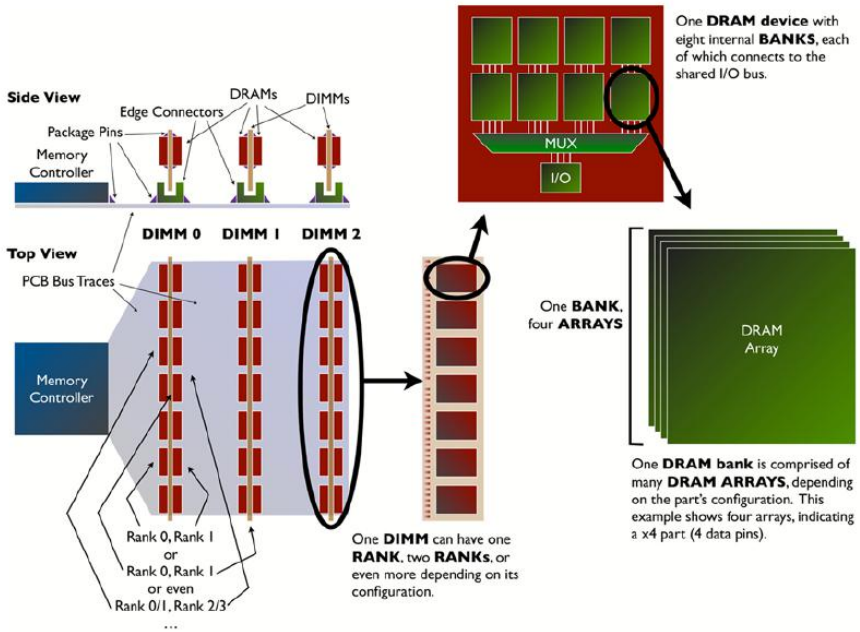


Figure 1.5: DIMMs, ranks, banks, and arrays. A system has potentially many DIMMs, each of which may contain one or more ranks. Each rank is a set of ganged DRAM devices, each of which has potentially many banks. Each bank has potentially many constituent arrays, depending on the parts data width.

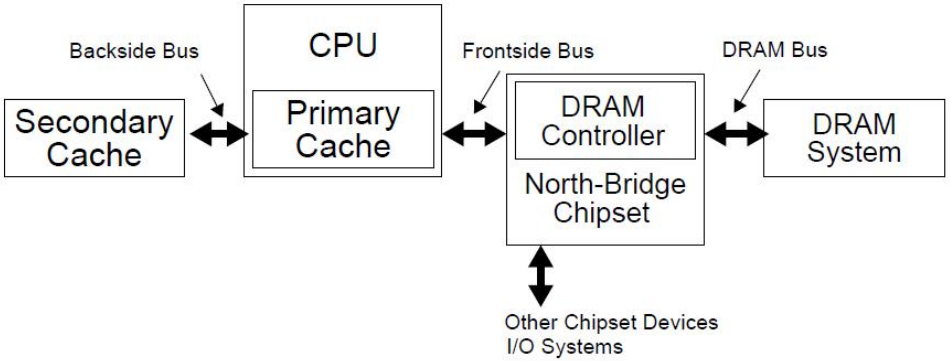


Figure 4.1: Memory System Architecture

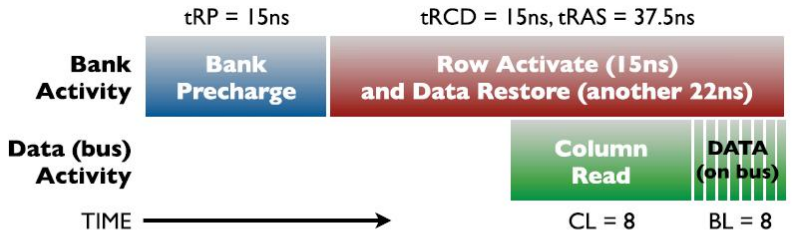


Figure 1.7: The costs of accessing DRAM. Before a DRAM cell can be read or written, the device must first be placed into a state that allows read/write access to the cells. This is a two step process: the **precharge** command initializes the banks sense amplifiers (sets the voltage on the bitlines appropriately), and the **activate** command reads the row data from the specified storage capacitors into the sense amplifiers. Once the sense amplifiers have read the data, it can be read/written by the memory controller. Precharge and activate together take several tens of nanoseconds; reading or writing takes roughly a nanosecond. Timing values taken from a Micron 2Gb DDR3-1066 part.

参考文献

Davis, B. T. (2001). Modern dram architectures, University of Michigan: 221.

Jacob, B. (2009). The Memory System, Morgan & Claypool.

Review: Virtual Memory and Caches

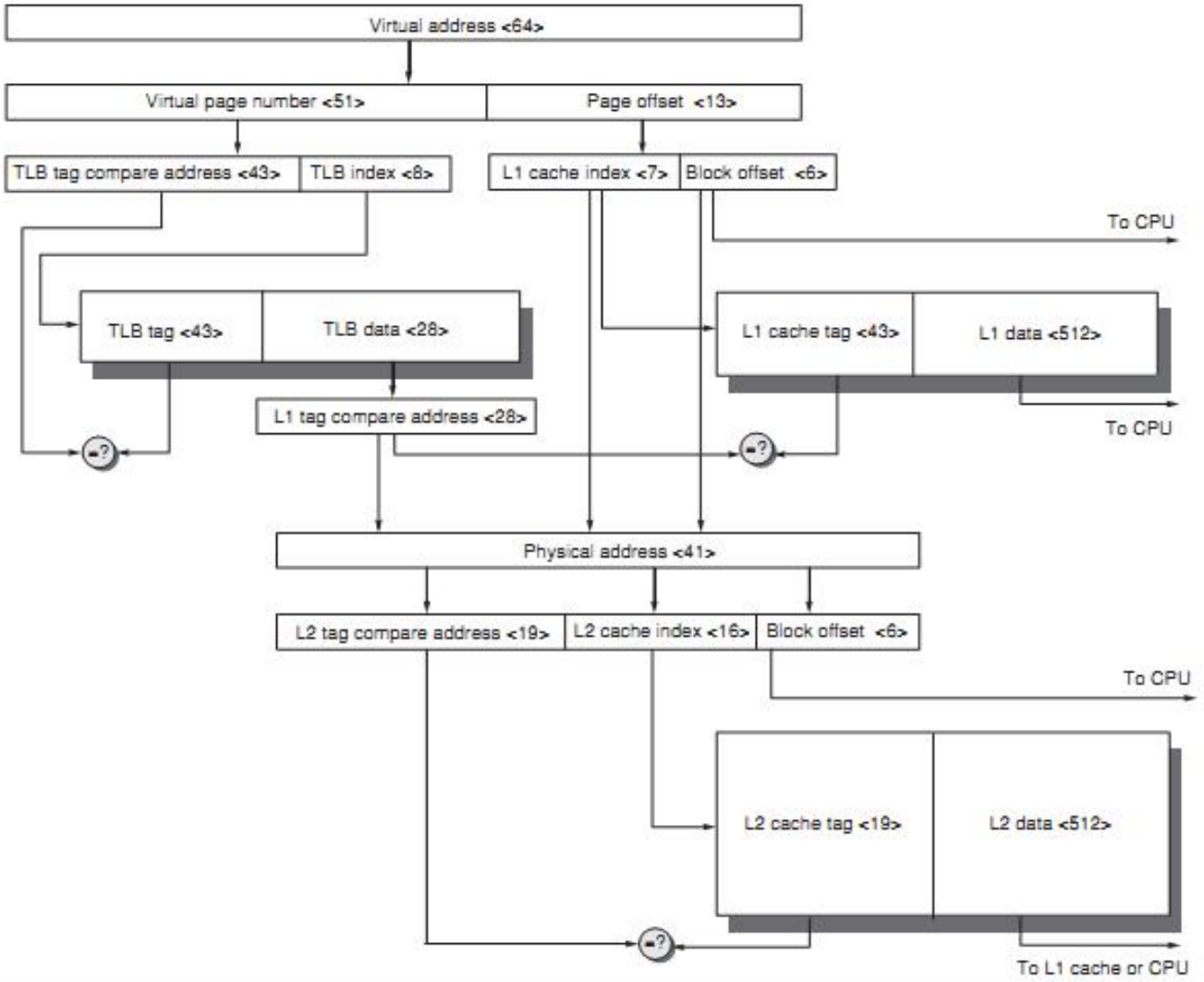


Figure C.24 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 8 KB. The TLB is direct mapped with 256 entries. The L1 cache is a direct-mapped 8 KB, and the L2 cache is a direct-mapped 4 MB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 41 bits. The primary difference between this simple figure and a real cache is replication of pieces of this figure.

第5章 指令级并行

5.1 指令级并行的基本概念及静态指令流调度

ILP及挑战性问题

软件方法挖掘指令集并行

5.2硬件方法挖掘指令级并行

5.2-1 指令流动态调度方法之一：Scoreboard

5.2-2 指令流动态调度方法之二：Tomasulo

5.3 分支预测方法

5.4 基于硬件的推测执行

5.5 存储器访问冲突消解及多发射技术

5.6 多线程技术

5.1 ILP的基本概念及挑战

ILP概述 → 循环展开 → 静态指令流调度

回顾：系统结构的Flynn分类 (1966)

- **SISD: Single instruction stream, single data stream**
  - 单处理器模式
- **SIMD: Single instruction stream, multiple data streams**
  - 相同的指令作用在不同的数据
  - 可用来挖掘数据级并行(Data Level Parallelism)
  - 如：Vector processors, SIMD instructions, and Graphics processing units
- **MISD: Multiple instruction streams, single data stream**
  - No commercial implementation
- **MIMD: Multiple instruction streams, multiple data streams**
  - 通用性最强的一种结构，可用来挖掘线程级并行、数据级并行.....
  - 组织方式可以是松耦合方式也可以是紧耦合方式

Recap：Levels of Parallelism

- **请求级并行**
  - 多个任务可被分配到多个计算机上并行执行
- **进程级并行**
  - 进程可被调度到不同的处理器上并行执行
- **线程级并行**
  - 任务被组织成多个线程，多个线程共享一个进程的地址空间
  - 每个线程有自己独立的程序计数器和寄存器文件
- **数据级并行**
  - 单线程（逻辑上）中并行处理多个数据 (SIMD/Vector execution)
  - 一个程序计数器, 多个执行部件
- **指令级并行**
  - 针对单一指令流，多个执行部件并行执行不同的指令

回顾: 基本流水线

- **流水线提高的是指令带宽（吞吐率），而不是单条指令的执行速度**
- **相关限制了流水线性能的发挥**
  - 结构相关：需要更多的硬件资源
  - 数据相关：需要定向，编译器调度
  - 控制相关：尽早检测条件，计算目标地址，延迟转移，预测
- **增加流水线的级数会增加相关产生的可能性**
- **异常，浮点运算使得流水线控制更加复杂**
- **编译器可降低数据相关和控制相关的开销**
  - Load 延迟槽
  - Branch 延迟槽
  - Branch预测

指令级并行的基本概念及挑战

- **ILP: 无关的指令重叠（并行）执行**
- **流水线的平均CPI**
- **Pipeline CPI = Ideal Pipeline CPI + Struct Stalls + RAW Stalls + WAR Stalls + WAW Stalls + Control Stalls + Memory Stalls**
- **本章研究：减少停顿（stalls)数的方法和技术**
- **基本途径**
  - 软件方法：
    - Gcc: 17%控制类指令，5 instructions + 1 branch；
    - 在基本块上，得到更多的并行性
    - 挖掘循环级并行
  - 硬件方法
    - 动态调度方法
  - 以MIPS的浮点数操作为例

采用的基本技术

Technique	Reduces	Section
Forwarding and bypassing	Potential data hazard stalls	A.2
Delayed branches and simple branch scheduling	Control hazard stalls	A.2
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences	A.8
Dynamic scheduling with renaming	Data hazard stalls and stalls from antidependences and output dependences	3.2
Dynamic branch prediction	Control stalls	3.4
Issuing multiple instructions per cycle	Ideal CPI	3.6
Speculation	Data hazard and control hazard stalls	3.5
Dynamic memory disambiguation	Data hazard stalls with memory	3.2, 3.7
Loop unrolling	Control hazard stalls	4.1
Basic compiler pipeline scheduling	Data hazard stalls	A.2, 4.1
Compiler dependence analysis	Ideal CPI, data hazard stalls	4.4
Software pipelining, trace scheduling	Ideal CPI, data hazard stalls	4,3
Compiler speculation	Ideal CPI, data, control stalls	4.4

本章遵循的指令延时

产生结果的指令	使用结果的指令	所需延时
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

- **(当使用结果的指令为BRANCH指令时除外)**

5.1 ILP的基本概念及挑战

循环展开+静态指令流调度

- **基本块的定义：**
  - 直线型代码，无分支；单入口；程序由分支语句连接基本块构成
- **循环级并行**
  - for (i = 1; i <= 1000; i++) x(i) = x(i) + s;
  - 计算x(i)时**没有数据相关**；可以并行产生1000个数据；
  - 问题：在生成代码时会有Branch指令 - **有控制相关**
  - 预测比较容易，但我们必须有预测方案
  - 向量处理机模型
    - load vectors x and y (up to some machine dependent max)
    - then do result-vec = xvec + yvec in a single instruction

简单循环及其对应的汇编程序

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

Loop:	LD	F0,0(R1)	;F0=vector element
	ADDD	F4,F0,F2	;add scalar from F2
	SD	0(R1),F4	;store result
	SUBI	R1,R1,8	;decrement pointer 8B (DW)
	BNEZ	R1,Loop	;branch R1!=zero
	NOP		;delayed branch slot

FP 循环中的相关

```
Loop:   LD      F0,0(R1)    ;F0=vector element
        ADDD    F4,F0,F2    ;add scalar from F2
        SD      0(R1),F4    ;store result
        SUBI    R1,R1,8     ;decrement pointer 8B (DW)
        BNEZ    R1,Loop     ;branch R1!=zero
        NOP                 ;delayed branch slot
```

产生结果的指令	使用结果的指令	所需延时
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

FP 循环中的Stalls

```
1 Loop:   LD      F0,0(R1)      ;F0=vector element
2         stall
3         ADDD    F4,F0,F2      ;add scalar in F2
4         stall
5         stall
6         SD      0(R1),F4      ;store result
7         SUBI    R1,R1,8       ;decrement pointer 8B (DW)
8         stall                 ;
9         BNEZ    R1,Loop       ;branch R1!=zero
10        stall                   ;delayed branch slot
```

产生结果的指令	*使用结果的指令*	*所需的延时*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Integer op	Integer op	0

10 clocks: 是否可以通过调整代码顺序使stalls减到最小

FP 循环中的最少Stalls数

```
1 Loop:  LD     F0,0(R1)
2        SUBI   R1,R1,8
3        ADDD   F4,F0,F2
4        stall
5        BNEZ   R1,Loop ;delayed branch
6        SD     8(R1),F4      ;altered when move past SUBI
```

Swap BNEZ and SD by changing address of SD

6 clocks: 通过循环展开4次是否可以提高性能?

循环展开4次(straight forward way)

```
1 Loop:  LD     F0,0(R1)    stall
2        ADDD   F4,F0,F2    stall    stall
3        SD     0(R1),F4              ;drop SUBI & BNEZ
4        LD     F6,-8(R1)   stall
5        ADDD   F8,F6,F2    stall stall
6        SD     -8(R1),F8             ;drop SUBI & BNEZ
7        LD     F10,-16(R1) stall
8        ADDD   F12,F10,F2  stall stall
9        SD     -16(R1),F12           ;drop SUBI & BNEZ
10       LD     F14,-24(R1)  stall
11       ADDD   F16,F14,F2   stall  stall
12       SD     -24(R1),F16
13       SUBI   R1,R1,#32   stall     ;alter to 4*8
14       BNEZ   R1,LOOP
15       NOP
```

Rewrite loop to minimize stalls?

15 + 4 x (1+2) + 1 = 28 cycles, or 7 per iteration

Assumes R1 is multiple of 4

Stalls数最小的循环展开

```
1 Loop:  LD     F0,0(R1)
2        LD     F6,-8(R1)
3        LD     F10,-16(R1)
4        LD     F14,-24(R1)
5        ADDD   F4,F0,F2
6        ADDD   F8,F6,F2
7        ADDD   F12,F10,F2
8        ADDD   F16,F14,F2
9        SD     0(R1),F4
10       SD     -8(R1),F8
11       SUBI   R1,R1,#32
12       SD     16(R1),F12
13       BNEZ   R1,LOOP
14       SD     8(R1),F16       ; 8-32 = -24
```

- 代码移动后
  - SD移动到SUBI后，注意偏移量的修改
  - Loads移动到SD前，注意偏移量的修改

14 clock cycles, or 3.5 per iteration

循环展开示例小结

- 循环展开对循环间无关的程序是有效降低stalls的手段(**循环级并行**）.
- **指令调度**，必须保证程序运行的结果不变
- 注意循环展开中的Load和Store,不同次循环的Load和Store 是相互独立的。需要分析对存储器的引用，保证他们没有引用同一地址.（**检测存储器访问冲突**）
- 不同次的循环，使用不同的寄存器 （**寄存器重命名**）
- 删除不必要的测试和分支后，需调整循环步长等控制循环的代码.（**步长调整**）
- 移动SD到SUBI和BNEZ后，需要调整SD中的偏移

5.1 ILP的基本概念及挑战

ILP概述 → 循环展开 → 静态指令流调度

从编译器角度看代码移动（1/4)

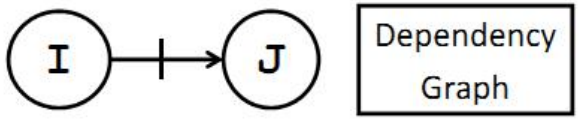
- **编译器分析程序的相关性依赖于给定的流水线**
- **编译器通过指令调度来消除相关**
- **(True) 数据相关（Data dependencies)**
  - 对于指令i和j，如果指令 j使用指令i产生的结果，或指令 j与指令 k相关，并且指令 k 与指令 i有数据相关.
- **如果相关，不能并行执行**

- **对于寄存器比较容易确定(fixed names)**
- **但对memory的引用，比较难确定:**
  - 100(R4) = 20(R6)?
  - 在不同次的循环中，20(R6) = 20(R6)?

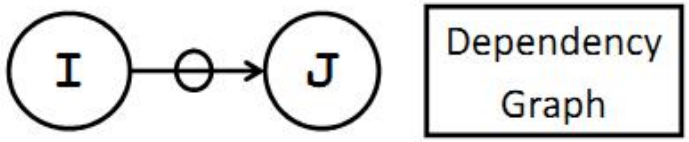
下列程序哪里有数据相关?

```
1 Loop:   LD     F0,0(R1)
2         ADDD   F4,F0,F2
3         SUBI   R1,R1,8
4         BNEZ   R1,Loop  ;delayed branch
5         SD     8(R1),F4  ;altered when move past SUBI
```

从编译器角度看代码移动(2/4)

- **另一种相关称为名相关（ name dependence）：两条指令使用同名参数(register or memory location)但不交换数据**
  - 反相关（Antidependence） (WAR if a hazard for HW)
    - Instruction j 所写的寄存器或存储单元，与 instruction i 所读的寄存器或存储单元相同，注instruction i 是先执行
  - 

  - 输出相关(Output dependence) (WAW if a hazard for HW)
    - Instruction i 和instruction j 对同一寄存器或存储单元进行写操作，必须保证两条指令的写顺序



下列是否有名相关?

```
1 Loop: LD    F0,0(R1)
2       ADDD  F4,F0,F2
3       SD    0(R1),F4      ;drop SUBI & BNEZ
4       LD    F0,-8(R1)
5       ADDD  F4,F0,F2
6       SD    -8(R1),F4     ;drop SUBI & BNEZ
7       LD    F0,-16(R1)
8       ADDD  F4,F0,F2
9       SD    -16(R1),F4    ;drop SUBI & BNEZ
10      LD    F0,-24(R1)
11      ADDD  F4,F0,F2
12      SD    -24(R1),F4
13      SUBI  R1,R1,#32     ;alter to 4*8
14      BNEZ  R1,LOOP
15      NOP
```

如何消除名相关?

下列是否存在名相关?

```
1 Loop:LD    F0,0(R1)
2      ADDD  F4,F0,F2
3      SD    0(R1),F4      ;drop SUBI & BNEZ
4      LD    F6,-8(R1)
5      ADDD  F8,F6,F2
6      SD    -8(R1),F8     ;drop SUBI & BNEZ
7      LD    F10,-16(R1)
8      ADDD  F12,F10,F2
9      SD    -16(R1),F12   ;drop SUBI & BNEZ
10     LD    F14,-24(R1)
11     ADDD  F16,F14,F2
12     SD    -24(R1),F16
13     SUBI  R1,R1,#32     ;alter to 4*8
14     BNEZ  R1,LOOP
15     NOP
```

这种方法称为寄存器重命名（register renaming）

从编译器角度看代码移动（3/4)

- **访问存储单元时，很难判断名相关**
  - 100(R4) = 20(R6)?
  - 不同次的循环，20(R6) = 20(R6)?
- **我们给出的示例要求编译器假设R1不变，因此:**

0(R1) ≠ -8(R1) ≠ -16(R1) ≠ -24(R1)

  - 因此loads和stores之间相互无关可以移动

从编译器角度看代码移动（4/4)

- **最后一种相关称为控制相关(control dependence)**

  Example:

  if p1 {S1;};

  if p2 {S2;};

  S1 依赖于P1的测试结果，S2依赖于P2的测试。

- **处理控制相关的原则：**
  - 受分支指令控制的指令，不能移到控制指令之前，以免该指令的执行不在分支指令的控制范围.
  - 不受分支指令控制的指令，不能移到控制指令之后，以免该指令的执行受分支指令的控制.
- **减少控制相关可以提高指令的并行性**

下列程序段的控制相关

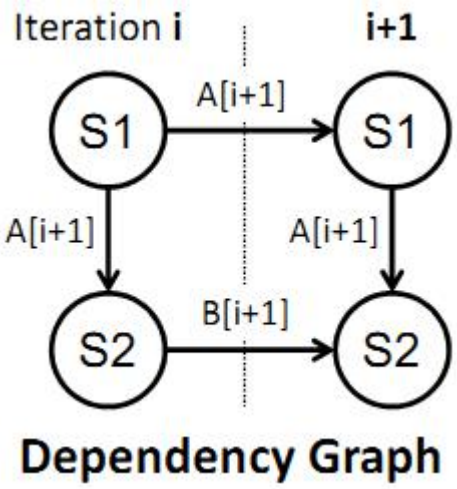
```
1 Loop:   LD     F0,0(R1)
2         ADDD   F4,F0,F2
3         SD     0(R1),F4
4         SUBI   R1,R1,8
5         BEQZ   R1,exit
6         LD     F0,0(R1)
7         ADDD   F4,F0,F2
8         SD     0(R1),F4
9         SUBI   R1,R1,8
10        BEQZ   R1,exit
```

```
11        LD     F0,0(R1)
12        ADDD   F4,F0,F2
13        SD     0(R1),F4
14        SUBI   R1,R1,8
15        BEQZ   R1,exit
....
```

循环间相关（1/4）

Example: 下列程序段存在哪些数据相关?
(A,B,C 指向不同的存储区且不存在覆盖区)

```
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 */
}
```



Dependency Graph

1. S2使用由S1在同一循环计算出的 A[i+1].
2. S1 依赖于前一次循环的S1；S2也依赖于前一次循环的S2。

这种存在于循环间的相关，称为 **"loop-carried dependence"**

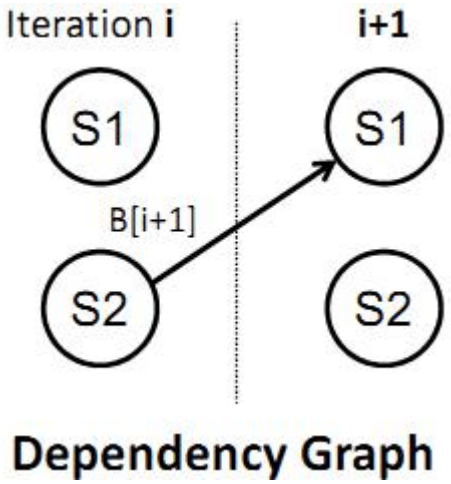
表示循环间存在相关，不能并行执行，它与我们前面的例子中循环间无关是有区别的

无回路的循环间相关（2/4)-

Example:A,B,C,D distinct & nonoverlapping

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

Non-Circular Loop-Carried Dependence



Dependency Graph

1. S1和S2没有相关，S1和S2互换不会影响程序的正确性
2. 在第一次循环中，S1依赖于前一次循环的B[1].
3. S1依赖上一次循环的S2，但S2不依赖S1

循环间相关（3/4）-循环变换

OLD:

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

NEW:

```
  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];
```

循环间相关（4/4）-Dependence Distance

通常循环间相关呈现为递推关系

```
for (i=1; i<N; i++) A[i] = A[i-1] + B[i];
```

相关的距离可能大于1

```
for (i=4; i<N; i++) A[i] = A[i-4] + B[i];
```

可以通过循环展开增加循环内的并行性

```
for (i=4; i<N; i=i+4)
{    A[i]  = A[i-4] + B[i];
     A[i+1] = A[i-3] + B[i+1];
     A[i+2] = A[i-2] + B[i+2];
     A[i+3] = A[i-1] + B[i+3];
 }
```

循环展开示例小结

- 循环展开对循环间无关的程序是有效降低stalls的手段(循环级并行）.
- 指令调度，必须保证程序运行的结果不变
- 注意循环展开中的Load和Store,不同次循环的Load和Store 是相互独立的。需要分析对存储器的引用，保证他们没有引用同一地址.（检测存储器访问冲突）
- 不同次的循环，使用不同的寄存器 （寄存器重命名）
- 删除不必要的测试和分支后，需调整循环步长等控制循环的代码.（步长调整）
- 移动SD到SUBI和BNEZ后，需要调整SD中的偏移

Summary

- **指令级并行(ILP)：流水线的平均CPI**
  - Pipeline CPI = Ideal Pipeline CPI + Struct Stalls + RAW Stalls + WAR Stalls + WAW Stalls + Control Stalls +……
  - 提高指令级并行的方法
    - 软件方法：指令流调度，循环展开，软件流水线，trace scheduling
    - 硬件方法
- **软件方法：指令流调度-循环展开**
    - 指令调度，必须保证程序运行的结果不变
    - 偏移量的修改
    - 寄存器的重命名
    - 循环步长的调整

Acknowledgements

- **These slides contain material developed and copyright by:**
  - John Kubiatowicz (UCB)
  - Krste Asanovic (UCB)
  - John Hennessy (Standford)and David Patterson (UCB)
  - Chenxi Zhang (Tongji)
  - Muhamed Mudawar (KFUPM)
- **UCB material derived from course CS152、CS252、CS61C**
- **KFUPM material derived from course COE501、COE502**