Decoupled Access/Execute Computer Architectures 阅读报告

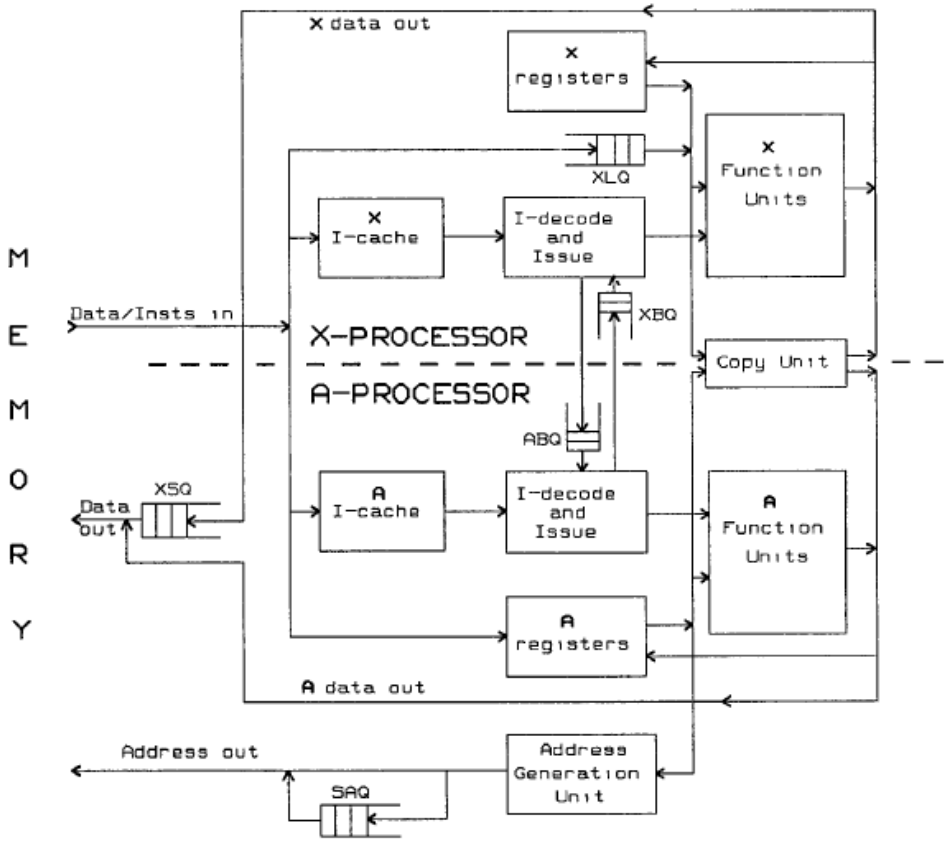
中国科学技术大学 齐寒

1. INTRODUCTION

当代高性能计算通过矢量运算进行了很大的提速，但在标量运算上收到限制，根据Amdahl's law.标量运算过慢最终会拖低机器性能，本文给出一种优化标量计算的体系结构，即分离访问/运算体系结构，并对其性能提升进行了证明，同时提出了预防死锁与死锁解决方法。

2. DECOUPLED ARCHITECTURE

2.0 OVERVIEW



1） X、A：独立寄存器、数目长度无需一致　　A-P：地址计算和读写　X-P：功能计算

2） 读取的数据进入 XLQ/A-REG，X 不直接读取数据，X 运算结束后需要存储的数据进入 XSQ，A 计算存储地址完成后进入 SAQ，当 XSQ 数据可用时与队头配对

3） XBQ、ABQ 用于处理分支，根据队列头的结果判断分支采取或不采取，为提高性能应该尽可能多由 A 确定分支，减少 X 的运算。

4） 例子分析，提速点在于 X 可以在地址计算的同时进行运算，而不必等待数据

```
      q = 0.0
      Do 1 = 1, 400
1     x(k) = q + y(k)*(r*z(k + 10) + t*z(k + 11))
```

(a)

```
      A4 ← −400          • negative loop count
      A2 ← 0             • initialize index (k)
      A3 ← 1             • index increment
      X2 ← r             • load loop invariants
      X5 ← t             •      into registers
loop: X3 ← z + 10, A2    • load z(k + 10)
      X7 ← z + 11, A2    • load z(k + 11)
      X4 ← X2*fX3        • r*z(k + 10) − floating multiply
      X8 ← X5*fX7        • t*z(k + 11)
      X7 ← y, A2         • load y(k)
      X6 ← X3 + fX4      • r*z(k + 10) + t*z(k + 11)
      A4 ← A4 + 1        • increment loop count
      X4 ← X7*fX6        • y(k)*(r*z(k + 10) + t*z(k + 11))
      A0 ← A4            • copy loop count for JAM
      x, A4 ← X4         • store into x(k)
      A2 ← A2 + A3       • increment index
      JAM loop           • branch if A0 < 0
```

(b)

```
                Access                          Execute
      A4 ← −400
      A2 ← 0
      A3 ← 1
      XLQ ← r                                   X2 ← XLQ
      XLQ ← t                                   X5 ← XLQ
loopα: XLQ ← z + 10, A2     loopx:              X4 ← X2*fXLQ
      XLQ ← z + 11, A2                          X3 ← X5*fXLQ
      XLQ ← y, A2                               X6 ← X3 + fX4
      A4 ← A4 + 1                               XSQ ← XLQ*fX6
      X, A2 ← XSQ                               BFQ loopx     • remove outcome from XBQ
      A0 ← A4
      A2 ← A2 + A3
      JAM loopα     • place outcome in XBQ
```

(c)

2.1 Memory Interlocks

1） 本文认为内存读取数据顺序与发出请求的顺序一致以便分析

2） 在数据可用之前发出存储指令可以减少管道堵塞，对性能有提升。

Q1:是否会存在存储访问乱序情况，例如 A 计算的地址匹配到错误的数据?

A1:顺序一致不会出现乱序情况，关于乱序情况本文未作讨论，可以通过约束避免

Q2: 写后读、读后写问题如何处理?

A2: 个人认为可以按照正常的添加 stall 进行处理，匹配前一直 stall，但这种处理方式比较牺牲性能。

Q3:如果存储到同一位置怎么办，即 WAW，且 A、X 是两个处理器，是否存在 “Write and Write “?

A3: 通过设定优先级，这种情况全部由 A 处理，X 进入 stall

2.2 Queue Architecture and Implementation

1） Xn-1 用于表示队头或队尾，方便队列操作数像寄存器一样被引用，访问队列不需要特殊的指令或寻址模式

2） 简化了对于满空条件的测试队列

Q4:这样做的好处?

A4: XLQ 可以实现为存储在寄存器文件中的标准循环缓冲区

2.3 Queue Timing

1）涉及 XLQ 的数据访问速度将与涉及 X 寄存器的数据访问速度一样快

Q5:具体论证不是很理解

A5:

2.4 Interrupts and Traps

1）与传统架构相比，队列的使用可能会存储更多的状态信息，但事实并非如此;分离的体系结构可能需要更少的通用寄存器。

2）CRAY-1 允许指令无序地完成，但不提供精确的陷阱。为方便比较，本文描述的解耦实现也做了相同的工作。

3）在解耦的体系结构中，与传统体系结构相比，虚拟内存的性能损失可能更小。

3. PERFORMANCE STUDY

Issue Time (*clock period*)		
0	loop:	$X_3 \leftarrow Z + 10, A_2$
2		$X_7 \leftarrow Z + 11, A_2$
11		$X_4 \leftarrow X_2 * f X_3$
13		$X_3 \leftarrow X_5 * f X_7$
14		$X_7 \leftarrow y, A_2$
20		$X_6 \leftarrow X_3 + f X_4$
21		$A_4 \leftarrow A_4 + 1$
26		$X_4 \leftarrow X_7 * f X_6$
27		$A_0 \leftarrow A_4$
33		$X, A_2 \leftarrow X_4$
35		$A_2 \leftarrow A_2 + A_3$
36		JAM loop

(a)

Issue Time		*Access*
0	loop*a*:	XLQ ← Z + 10, A_2
2		XLQ ← Z + 11, A_2
4		XLQ ← *y*, A_2
6		$A_4 \leftarrow A_4 + 1$
7		X, A_2 ← XSQ
9		$A_0 \leftarrow A_4$
10		$A_2 \leftarrow A_2 + A_3$
11		JAM loop*a*

(b)

Issue Time		*Execute*
0	loopx:	$X_4 \leftarrow X_2 * fXLQ$
1		$X_3 \leftarrow X_5 * fXLQ$
8		$X_6 \leftarrow X_3 + fX_4$
14		$XSQ \leftarrow XLQ * fX_6$
15		BFQ loop*x*

(c)

1) 测试使用 14 个 Lawrence Livermore 循环，使用 CFT 生成的实际目标代码（关闭了矢量化），代表了程序的混合，有些是可向量化的，有些不是。
2) 所有的时间通过 CRAY-1 模拟器和威斯康星大学开发的解耦架构模拟器完成
3) CRAY-1 每次通过循环需要 41 个时钟周期(通过循环需要 36 个时钟周期，再加上下面的分支需要 5 个时钟周期)
4) a 处理器可以使每个循环在 16 个时钟周期内通过(包括所取分支的 5 个时钟周期)。x 处理器需要 20 个时钟周期，会落后于 a 处理器。然而，在前两个循环(x 处理器等待 XLQ)通过之后，计算以稳定的速度进行，每次迭代有 20 个时钟周期——略高于单流版本的两倍。

Table I. Performance Simulation Results for the 14 Lawrence Livermore Loops

Loop	CRAY-1	Decoupled	Speedup
1	41	20	2.1
2	60	32	1.9
3	26	14	1.9
*4	38	20	1.9
*5	62	48	1.3
*6	64	47	1.4
7	81	56	1.4
*8	200	118	1.7
9	96	58	1.7
10	101	63	1.6
*11	27	21	1.3
12	27	17	1.6
*13	146	135	1.1
*14	149	126	1.2
Average			1.58

[a] All times are in clock periods per loop iteration. Loops that cannot be vectorized by the CFT compiler are marked with an asterisk (*).

5) 加速率是通过将 CRAY-1 时钟周期除以去耦架构时钟周期计算，在 1.1 和 2.1 之间变化。平均加速率是 1.58。对于 7 个不可向量化循环，平均加速为 1.41。

6) 由于循环迭代是独立的，并且访问/执行解耦是完整的，所以对于可向量化的循环，会出现更高的速度。循环 13 和 14 显示最少的加速。这是因为数据需要从 X 传递到 A。这严重限制了 A 在需要时提前运行和获取数据的自由。

4. SINGLE INSTRUCTION STREAM ARCHITECTURES

1）双指令流解耦的体系结构在概念上很简单，但存在两个缺点，必须处理两个交互的指令流，需要两个独立的指令缓存和解码/发行单元。这种硬件成本问题可以通过对两个指令获取/解码单元重复相同的设计来部分减轻

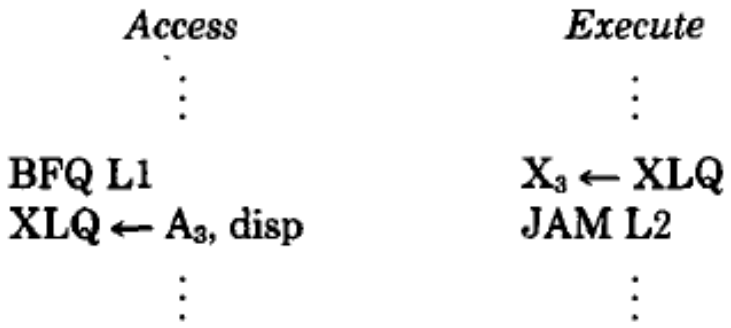
2）两种解决方式：

将两个指令流物理合并为一个；

出于编程和编译的目的，在概念上合并两个指令流，但是在程序加载到内存之前物理上分离指令流。

5. DEADLOCK

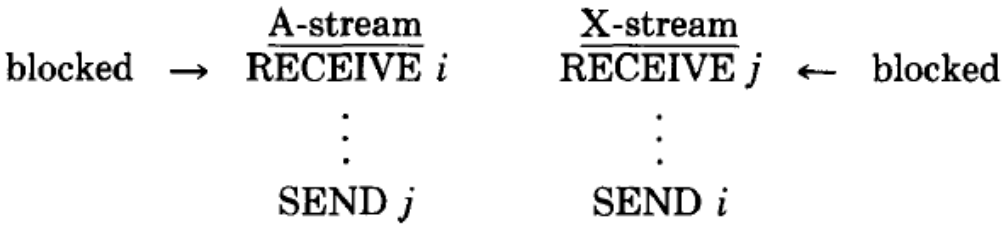
1) 死锁发生的情况：如果指令需要来自队列的数据(或分支结果)而队列是空的，或者需要将数据(或分支结果)放置到队列中且队列是满的(包括复制队列)，则可以阻止其发出指令。如果两个指令流同时由于上述原因被阻塞，就会发生死锁。

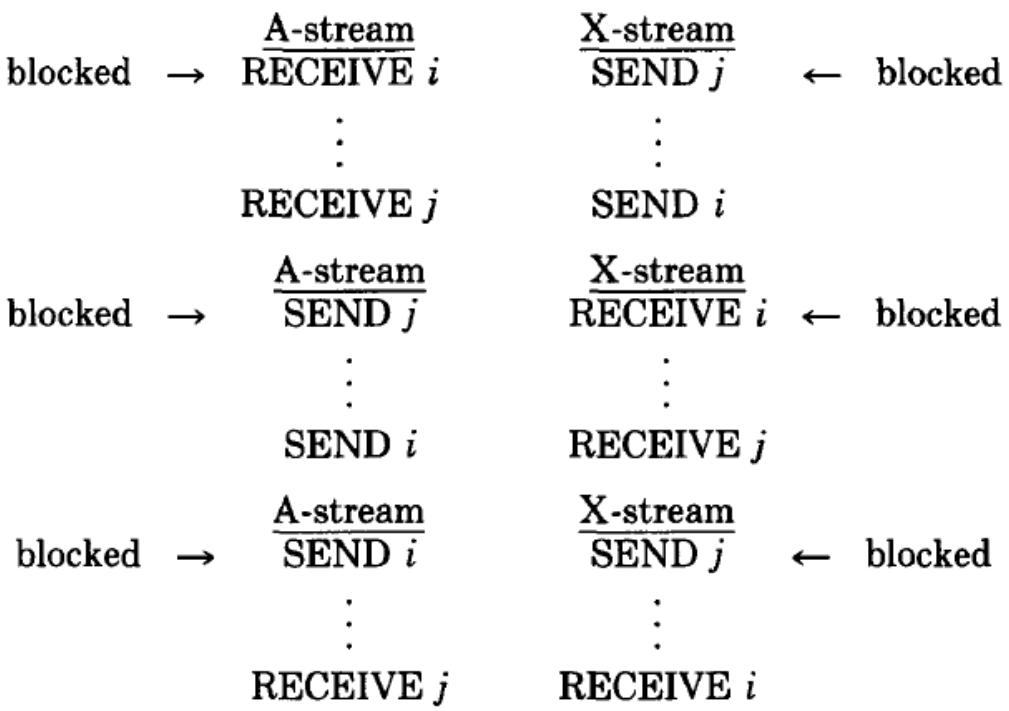


2）给出了无死锁操作的充分条件：

如果 A 和 X 处理器指令流可以正确地交叉，那么就不会发生死锁，证明如下：

死锁发生在由于队列为空或满而阻塞两个指令流时。更具体地说，如果队列已满，发送指令可以被阻塞;如果队列为空，接收指令可以被阻塞。四种可能死锁的情况如下：





两个流都被“等待接收的空队列”阻塞，而匹配的发送在流中稍后发生，然而，可以看到，这两种流的正确交错是不可能的。

6. CONCLUSIONS

1）允许高速指令发放;每个时钟周期最多可发出两个指令。这是包含 Flynn 瓶颈的传统单指令流处理器速率的两倍，也就是说，在指令获取/解码路径中，指令每时钟周期只能通过一个瓶颈。

2）它允许一种受限类型的无序指令问题，允许数据访问操作在需要之前就被很好地完成，但是不需要使用过去用于无序问题的复杂方法。

3）可以平滑(长时间)不可预测的内存访问延迟，并可能隐藏。在现代多处理器系统中，这一点尤为重要，因为内存带宽是一种关键资源。