# Static Scheduling and VLIW

## VLIW

产生的原因：超标量处理器的控制逻辑规模在不断增长

- 设 issue 宽度为 $W$，指令的生命周期为 $L$
- 每个 issue 的指令要与之前 $W \times L$ 条指令进行 check，则硬件增长的规模 $\propto W \times W \times L$
- 随着 $W$ 的增加，instruction window 会增大，$L$ 也会随之增大
- 控制逻辑的增长速度是大于 $O(W^2)$ 的

VLIW (Very Long Instruction Word) 是多个独立的指令由编译器打包而成的指令。区分 SIMD，VLIW 的指令是逻辑上无联系的。

由编译器寻找无关的指令并打包，一个 bundle 中的指令静态对齐，被送入 function unit

VLIW Philosophy：比较类似 RISC：简单的指令，简单的硬件。主要由编译器实现复杂的部分

## VLIW Basics

多个指令打成一个 bundle

VLIW 中每个 slot 对应一种固定的功能（整数，浮点，内存，……），指令的延时是固定的常数

架构需要得到如下保证

- 不需要指令内部的 RAW 检查，即指令本身可以并行
- 不需要等待数据，没有 data interlock

## VLIW Scheduling Model

Equals (EQ) Scheduling Model

- 每个指令的延时固定
- 高效的寄存器使用
- 不需要寄存器重命名或者 buffering，直接从 function unit 的出口 bypass 到入口，function unit 完成了直接写寄存器

Less-Than-or-Equals (LEQ) Scheduling Model

- 每个指令的延时小于等于一个固定的时间
  - 在 issue 之后的任意时间都可能写寄存器
  - 依赖的指令必须等待固定的延时
- 简化了精确中断

# Static Scheduling

## Challenges for Compilers

VLIW 编译器需要做到

- 调度指令使得并行度最大化
- 保证指令内部的并行性，即 bundle 内的指令无依赖
- 避免 data hazard

主要问题是

- 如何找到独立的指令
- 如何更大程度地优化（e. g. 消除公共子表达式，常量传播，消除死代码……）
- 如何提升指令的 fetch 率

答案：更大的基本块

- loop unrolling
- predicated execution

或者是优化更经常执行的路径

- trace
- superblock

## Loop Execution

考虑简单的循环

```
1 for (i = 0; i < N; i++)
2     B[i] = A[i] + C;
```

loop unrolling：将循环展开，增大基本块

software pipelining：展开后可以认为基本块由读取，计算，写回三部分组成，在进行计算时已经可以进行下一步循环的读取

software pipelining 可能需要更多寄存器

## Get More Room for Scheduling

basic block：只有一个进入点和一个离开点的代码块，控制流在基本块内部不会变化

atomic block：或是全部完成或是全部不完成的代码块

- 对于大部分 ISA 来说原子块对应的就是一条指令
- 如果执行中没有异常/中断/副作用，则一个基本块也可以看作一个原子块
- 只要保证真正的数据依赖（RAW），其余指令可以自由调整位置

一个基本块内部的 ILP 程度是有限的，为了增大 ILP，需要考虑多个基本块：寻找经常被执行的 block

- trace
- superblock
- hyperblock

## Safety and Legality in Code Motion

Safety：是否发生异常

Legality：是否结果总是正确

- illegal 的情况：将分支内的赋值操作提前，而另一个分支会用到原本的值
- unsafe 的情况：将 load 提前，提前后可能 load 会发生异常，产生新的异常

移动代码有如下限制

- 向下：将代码从基本块移到其某个目的基本块时
  - 其他目的基本块仍然能使用这个操作产生的结果
  - 不应该影响目的基本块的其他源基本块
- 向上：将某个指令移到其某个源基本块
  - 被其他目的基本块需求的值不应当被破坏
  - 不能产生新的异常

## Trace Scheduling

trace 是控制流图中经常执行的一条路径（多个入口，多个出口）

基本思路是在 trace 中寻找独立的指令打包成 VLIW

- 使用 profiling 决定 trace
- trace 途中会有进出口
- 忽略途中的出入
- 确定后需要为非 trace 的代码加入 fix-up/bookkeeping

选择 trace 的过程是

- 选择频率最高的基本块作为 seed BB

- 根据频率最高的边来扩展 trace
- 不要跨越 loop back edge
- 限定 trace 的长度

获取 trace 后再做 scheduling，并且保证语义正确（需要构建依赖图）

list scheduling

- 为每个指令添加优先级
- 初始化一个 ready list 包含所有数据就绪的指令
- 选择 ready list 中优先级最高的
- 将该指令加入 scheduling
- 将其余满足的指令再加入 ready list

> 指令的优先级可以以启发式规则生成
>
> - 依赖图中后继的数量
> - 从根节点开始的最大延时
> - 指令延时长度
> - 路径的 rank

补偿代码的实例参见第 10 周 ppt

tradeoff

- 优点
  - 寻找更多独立指令，减少 VLIW 中的 nop
- 缺点
  - 需要 profile
  - 需要额外的 fix-up，不常执行的路径会干涉常执行的路径
  - 有效性是基于 branch 的 bias，如果是 unbiased 的 branch 会减小 trace

## Superblock Scheduling

与 trace 的不同之处在于，superblock 是单入口多出口的，这样不常执行的路径不会干涉常执行的路径

tradeoff

- 优点
  - 比 trace 更多的优化机会
  - 减少了 bookkeeping
- 缺点
  - 仍然依赖于 profiling
  - 如果有 unbiased 的 branch 同样会减小 superblock 大小
  - 由于出口的存在，仍然需要额外的 fix-up

### Hyperblock Scheduling

使用 predicated execution 来消除 unbiased branch，增大 superblock 大小

hyperblock：单入口多出口，内部没有控制流

tradeoff

- 优点
  - 减少了 unbiased branch 的影响
- 缺点
  - 需要 ISA 支持 predicated execution
  - 所有 predicated execution 的缺点

## Remaining Issues for VLIW

Object-code compatibility：需要为每台机器重新编译代码

Object-code size：用于 padding 的 nop 增大代码体积，同理 loop unrolling/software pipelining

Scheduling variable latency memory operations：静态调度没办法解决 cache/memory 的冲突/miss

Knowing branch probabilities：对 profiling 有较高要求

Scheduling for statically unpredictable branches：优化大程度依赖于 branch 的情况

Precise interrupts can be challenging：一个 bundle 中一条指令发生异常如何处理

# IA-64

EPIC (Explicitly Parallel Instruction Computing)：显式并行指令计算

一个 VLIW 的架构

- group：一系列能安全并行的指令，通过 stop bits 标记
- bundle：一个 bundle 3 条指令，一个 group 可以分为多个 bundle

每个 bundle 中有 template bits，用于区分 group，以及为指令的 function unit 提供路由信息

## Non-Faulting loads

ld.s 是 speculative load，如果发生异常不处理，之后使用 r1 前需要 chk.s

如果 `ld.s r1 = [a]` 没有发生异常，`chk.s r1` 就是 nop，否则额外执行一句 `ld r1 = [a]`

这种 speculative 可以传递，如果在 ld.s 后直接使用了 r1：`use = r1`，则 `chk.s use` 会检查整个流是否发生异常，如果发生则执行 `ld r1 = [a]; use = r1`

使用了 speculative 结果的指令本身会变成 speculative，即屏蔽异常

# Data Speculation

可以更激进地 load

在最前面加一句 ld.a，原本 ld 的位置用 ld.c 代替，如果 ld.a 之后的 store 没有 alias，则 ld.c 为 nop，否则重新 load 一次

有一个 ALAT (Advanced Load Address Table)，ld.a 会在其中添加一个 entry，如果有 store 发生了 alias，则从中移除该条目，chk 或是 ld.c 指令会检查 ALAT，如果找不到条目则执行恢复的代码

# Rotating Register File

IA-64 有 128 个 GPR，rotating 是一种简单的 register renaming

由于调度循环需要大量寄存器，为每个 iteration 分配一个寄存器的集合

- RRB (Rotating Register Base) register 指向当前集合的 base
- 逻辑寄存器号 + RRB 就是当前物理寄存器
- 一般会将寄存器分为 rotating 和 non-rotating

# VLIW tradeoff

优点

- 不需要动态调度的硬件，简化设计
- 不需要 VLIW 内部的依赖检测，简化多指令 issue 的硬件，不需要寄存器重命名
- 不需要指令对齐，简化硬件

缺点

- 编译器更复杂，需要寻找独立指令填满 VLIW
- 当 issue width，指令延时，function unit 变化时需要重新编译
- 如果一条指令 stall，整个 bundle 都会停住