VLIW调度器的应用实践

部门：
作者：徐若玢
日期：2024.12

Security Level:

HUAWEI

Instruction-Level Parallelism

- Pipelining become universal technique in 1985
  - Overlaps execution of instructions
  - Exploits “Instruction Level Parallelism”
- Beyond this, there are two main approaches:
  - Hardware-based dynamic approaches
    - Used in server and desktop processors
    - Not used as extensively in PMP processors
  - Compiler-based static approaches
    - Not as successful outside of scientific applications

- When exploiting instruction-level parallelism, goal is to minimize CPI
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls

VLIW (very long instruction word) processors

VLIW Processors

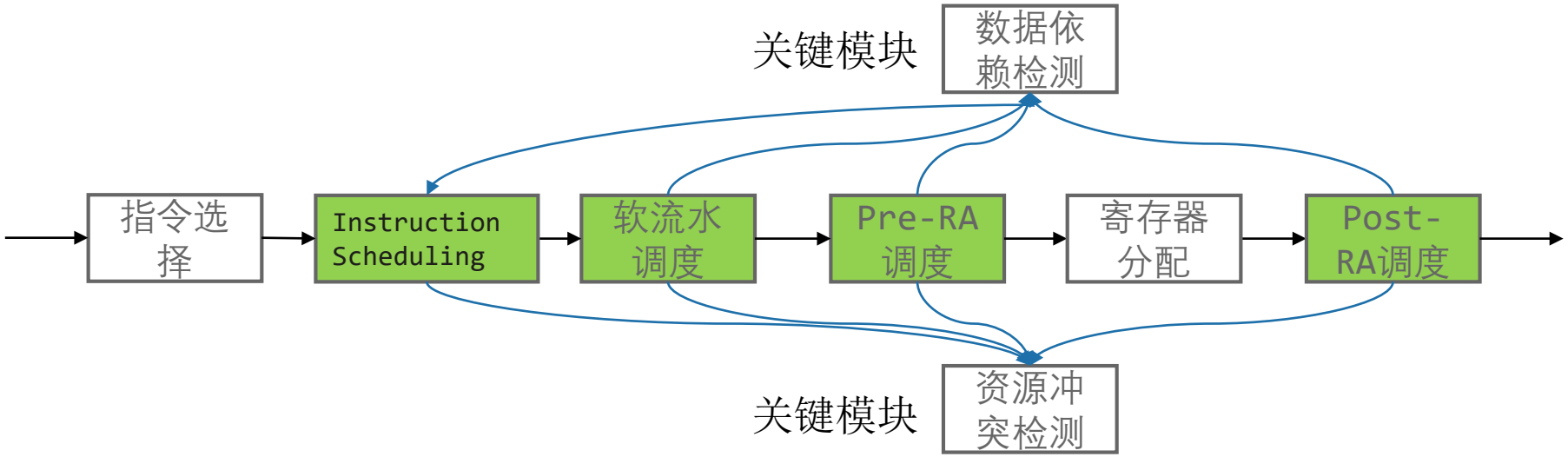
- Package multiple operations into one instruction
- Example VLIW processor:
  - One integer instruction (or branch)
  - Two independent floating-point operations
  - Two independent memory references
- Must be enough parallelism in code to fill the available slots

VLIW Processors

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
fld f0,0(x1)	fld f6,-8(x1)			
fld f10,-16(x1)	fld f14,-24(x1)			
fld f18,-32(x1)	fld f22,-40(x1)	fadd.d f4,f0,f2	fadd.d f8,f6,f2	
fld f26,-48(x1)		fadd.d f12,f0,f2	fadd.d f16,f14,f2	
		fadd.d f20,f18,f2	fadd.d f24,f22,f2	
fsd f4,0(x1)	fsd f8,-8(x1)	fadd.d f28,f26,f24		
fsd f12,-16(x1)	fsd f16,-24(x1)			addi x1,x1,-56
fsd f20,24(x1)	fsd f24,16(x1)			
fsd f28,8(x1)				bne x1,x2,Loop

- Disadvantages:
  - Statically finding parallelism
  - Code size
  - No hazard detection hardware
  - Binary code compatibility

贯穿后端全流程的多层次指令调度技术

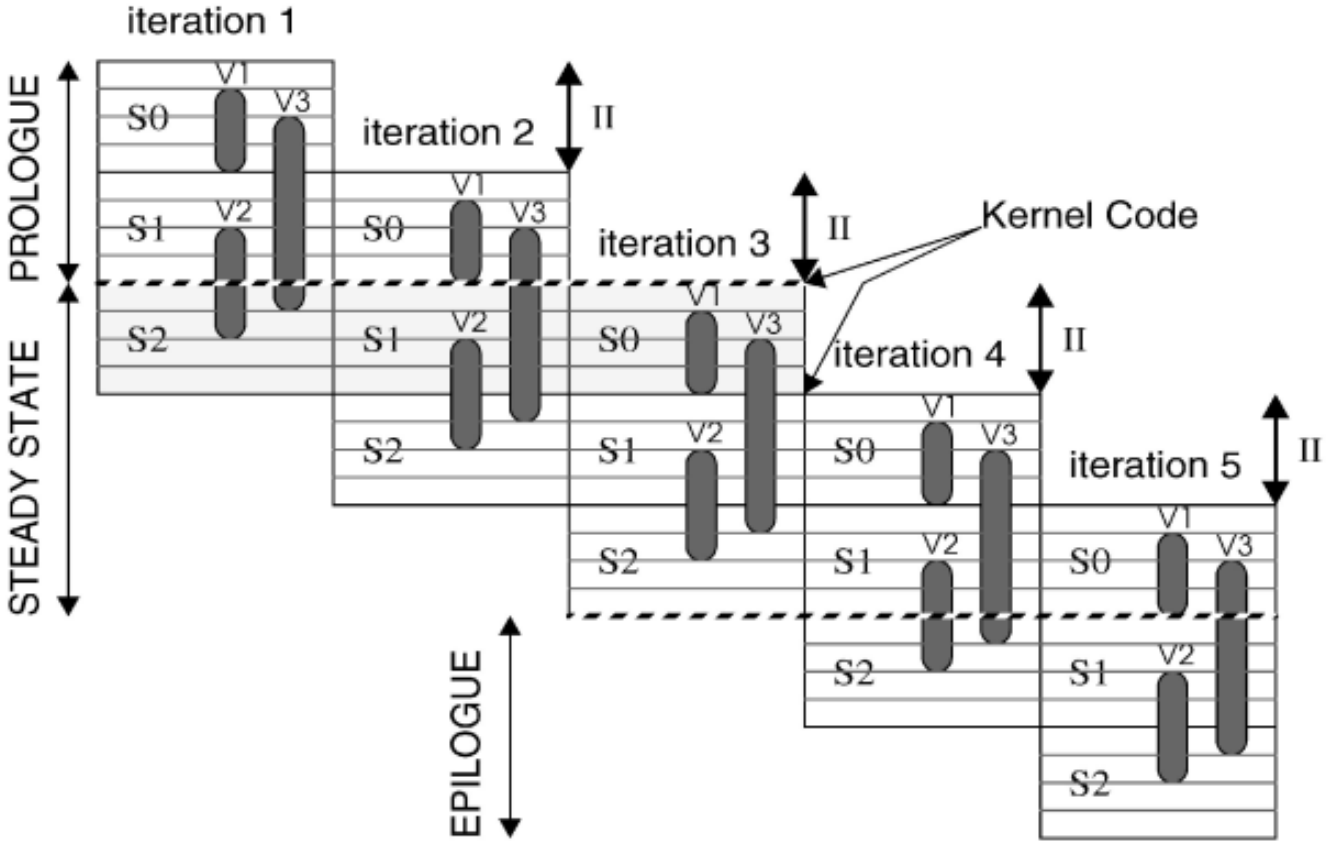


多轮调度，提供编译器在不同场景下的性能泛化能力。

1. Instruction Scheduling，基于SDNode的简单的调度器。
2. 软流水调度，优化循环，为benchmark最热代码提供最极致的性能。
3. Pre-RA调度，主要优化非循环，缩短变量的生命周期。也减轻寄存器压力，提高寄存器可重用度。
4. Post-RA调度（包含Packetizer），优化指令顺序，减少资源冲突，充分考虑硬件架构特性，提高指令并行性，提高程序性能。

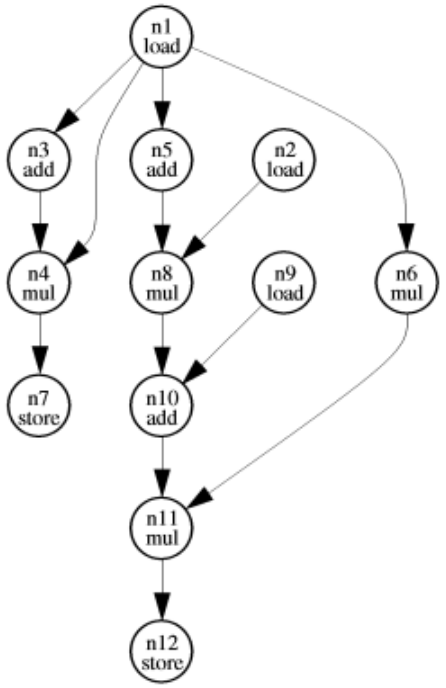
软流水调度

作用：软流水调度为模调度，优化循环，为benchmark最热代码提供最极致的性能。通过将不同迭代的指令折叠进同一迭代，减少stall，提升循环性能。



以上述循环为例，如果不使用软流水技术，每次迭代12 cycles，N次迭代共12*N cycles。如果使用软流水，N次迭代4*(N-2)+8*2 =4*N+8 cycles。大幅提高了循环的IPC。

软流水调度—示例



Hardware configuration:
1 add unit
1 mul unit
2 load/store units

Latencies:
add: 2 cycles
mul: 2 cycles
load: 2 cycles
store: 1 cycle

Fig. 2. Dependence graph for the motivating example.

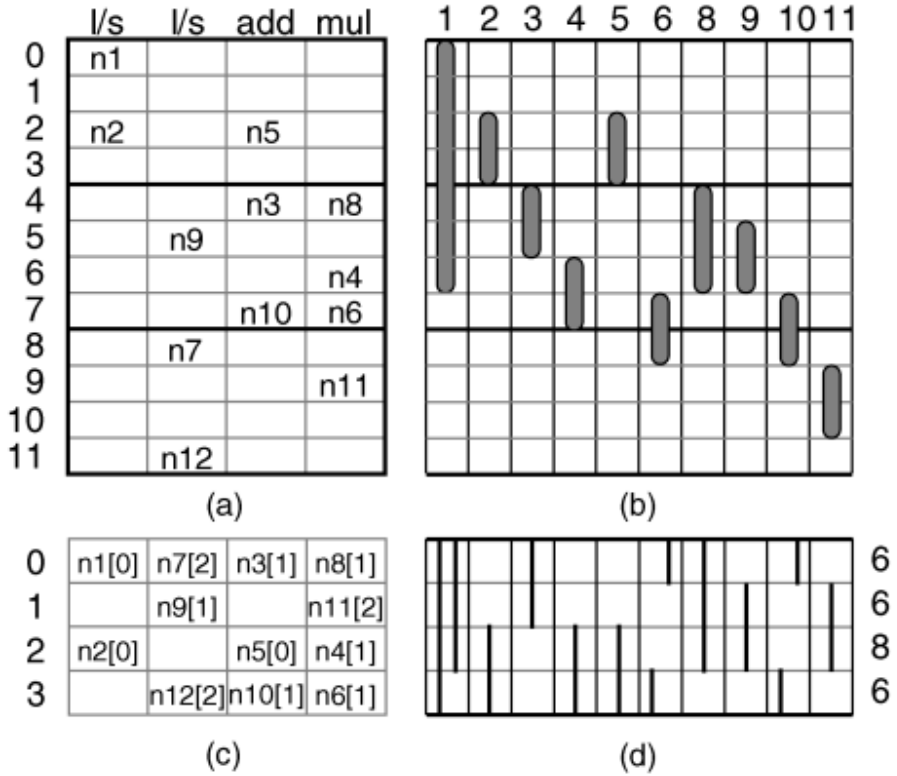


Fig. 5. SMS scheduling: (a) schedule of one iteration, (b) lifetimes of variables, (c) kernel of the schedule, and (d) register requirements.

注：图(c)中“[N]“表示stage，可以简单理解为第N次迭代的操作。比如n5[0]表示此时执行的是第0次迭代的n1, n8[1]表示执行的是第1次迭代的n8, 这个n8的输入来自于上一迭代的n5，对应于cycle(-2).

HUAWEI

软流水调度--LLVM软流水实现

```
563  /// We override the schedule function in ScheduleDAGInstrs to implement the
564  /// scheduling part of the Swing Modulo Scheduling algorithm.
565  void SwingSchedulerDAG::schedule() {
566    AliasAnalysis *AA = &Pass.getAnalysis<AAResultsWrapperPass>().getAAResults();
567    buildSchedGraph(AA);
568    addLoopCarriedDependences(AA);
569    updatePhiDependences();
570    Topo.InitDAGTopologicalSorting();
571    changeDependences();
572    postProcessDAG();
573    LLVM_DEBUG(dump());
```

1. 构建调度依赖图

```
575  NodeSetType NodeSets;
576  findCircuits(NodeSets);
577  NodeSetType Circuits = NodeSets;
578
579  // Calculate the MII.
580  unsigned ResMII = calculateResMII();
581  unsigned RecMII = calculateRecMII(NodeSets);
582
583  fuseRecs(NodeSets);
584
585  // This flag is used for testing and can cause correctness problems.
586  if (SwpIgnoreRecMII)
587    RecMII = 0;
588
589  setMII(ResMII, RecMII);
590  setMAX_II();
```

2. 计算RecII和ResII，MaxII=max(RecII, ResII)

```
623    computeNodeFunctions(NodeSets);
624
625    registerPressureFilter(NodeSets);
626
627    colocateNodeSets(NodeSets);
628
629    checkNodeSets(NodeSets);
630
631    LLVM_DEBUG({
632      for (auto &I : NodeSets) {
633        dbgs() << "  Rec NodeSet ";
634        I.dump();
635      }
636    });
637
638    llvm::stable_sort(NodeSets, std::greater<NodeSet>());
639
640    groupRemainingNodes(NodeSets);
641
642    removeDuplicateNodes(NodeSets);
643
644    LLVM_DEBUG({
645      for (auto &I : NodeSets) {
646        dbgs() << "  NodeSet ";
647        I.dump();
648      }
649    });
650
651    computeNodeOrder(NodeSets);
```

3. 计算调度顺序

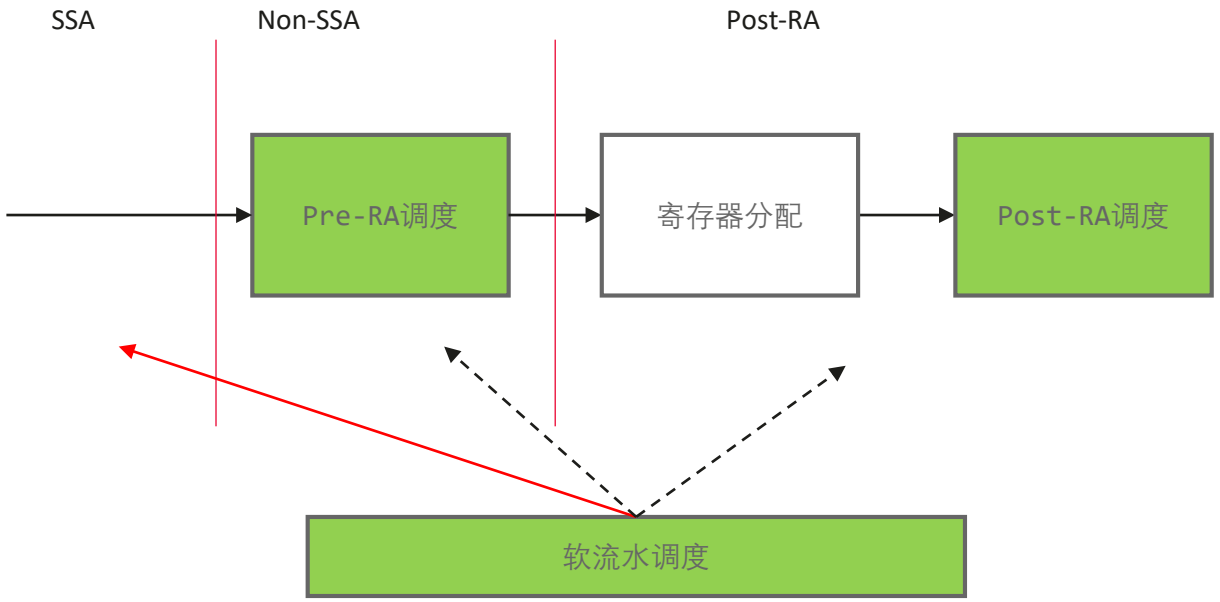
```
SMSchedule Schedule(Pass.MF, this);
Scheduled = schedulePipeline(Schedule);
```

4. 调度

```
// The experimental code generator can't work if there are InstChanges.
if (ExperimentalCodeGen && NewInstrChanges.empty()) {
  PeelingModuloScheduleExpander MSE(MF, MS, &LIS);
  MSE.expand();
} else if (MVECodeGen && NewInstrChanges.empty() &&
           LoopPipelinerInfo->isMVEExpanderSupported() &&
           ModuloScheduleExpanderMVE::canApply(Loop)) {
  ModuloScheduleExpanderMVE MSE(MF, MS, LIS);
  MSE.expand();
} else {
  ModuloScheduleExpander MSE(MF, MS, LIS, std::move(NewInstrChanges));
  MSE.expand();
  MSE.cleanup();
}
```

5. 生成prolog/kernel/epilog

软流水调度--下一步思考



1. 新增软流水算法，比如IMS（iterative modulo scheduling），在IMS和SMS最终的调度结果中择优
2. 尝试修改软流水位置，如左图，

（1）SSA的软流水更容易做数据流分析，但由于COPY/PHI的存在，不准确；

（2）Non-SSA无COPY/PHI，更准确；

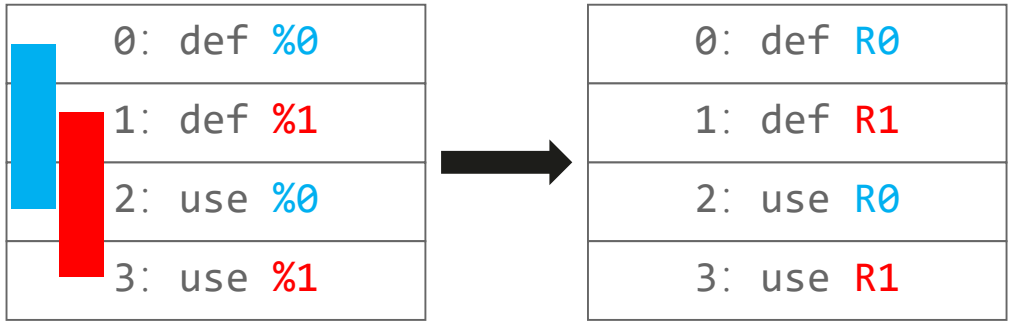
（3）Post-RA寄存器确定，相比前两者更准确，但是RA引入了伪依赖，效果可能不好。

可以结合回溯能力择优。

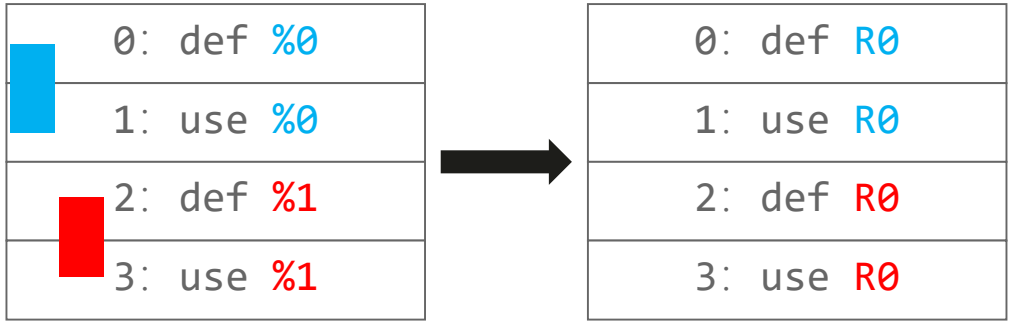
Pre-RA调度

Pre-RA调度在llvm里有两个，一个位于指令选择刚结束的Instruction Scheduling，一个位于Non-SSA阶段，后者称为MachineScheduler，我们讨论的主要是后者。

Pre-RA调度为表调度，主要优化非循环，缩短变量的生命周期。也减轻寄存器压力，提高寄存器可重用度。

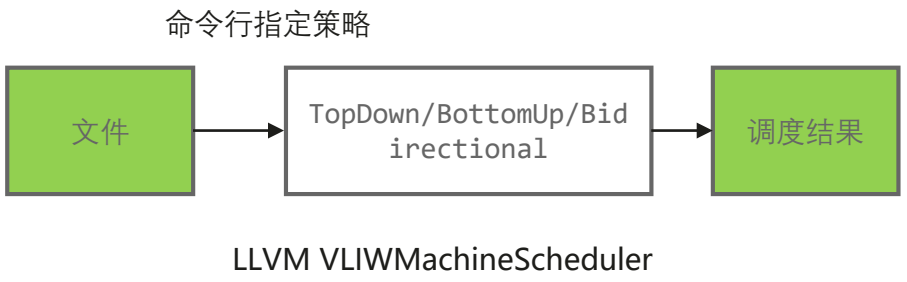


without Pre-RA scheduling

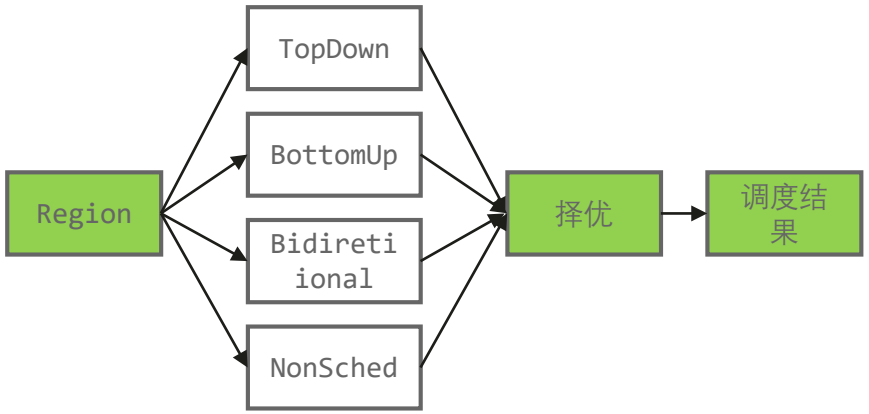


with Pre-RA scheduling

Pre-RA调度—LLVM的实现与改进



LLVM VLIWMachineScheduler



New MachineScheduler

TopDown：从Region（通常是BB）的入度为0的节点开始调度，当一个节点完成时，调度其后继节点。最终结果通常表现为BB的开头排布紧密，末尾排布稀疏。

BottomUp：从Region（通常是BB）的出度为0的节点开始调度，当一个节点完成时，调度其前驱节点。最终结果通常表现为BB的开头排布稀疏，末尾排布紧密。

Bidirectional：结合TopDown和BottomUp的特点，由算法自动选择下一个排布的节点在上方还是下方。

NonSched：不调度

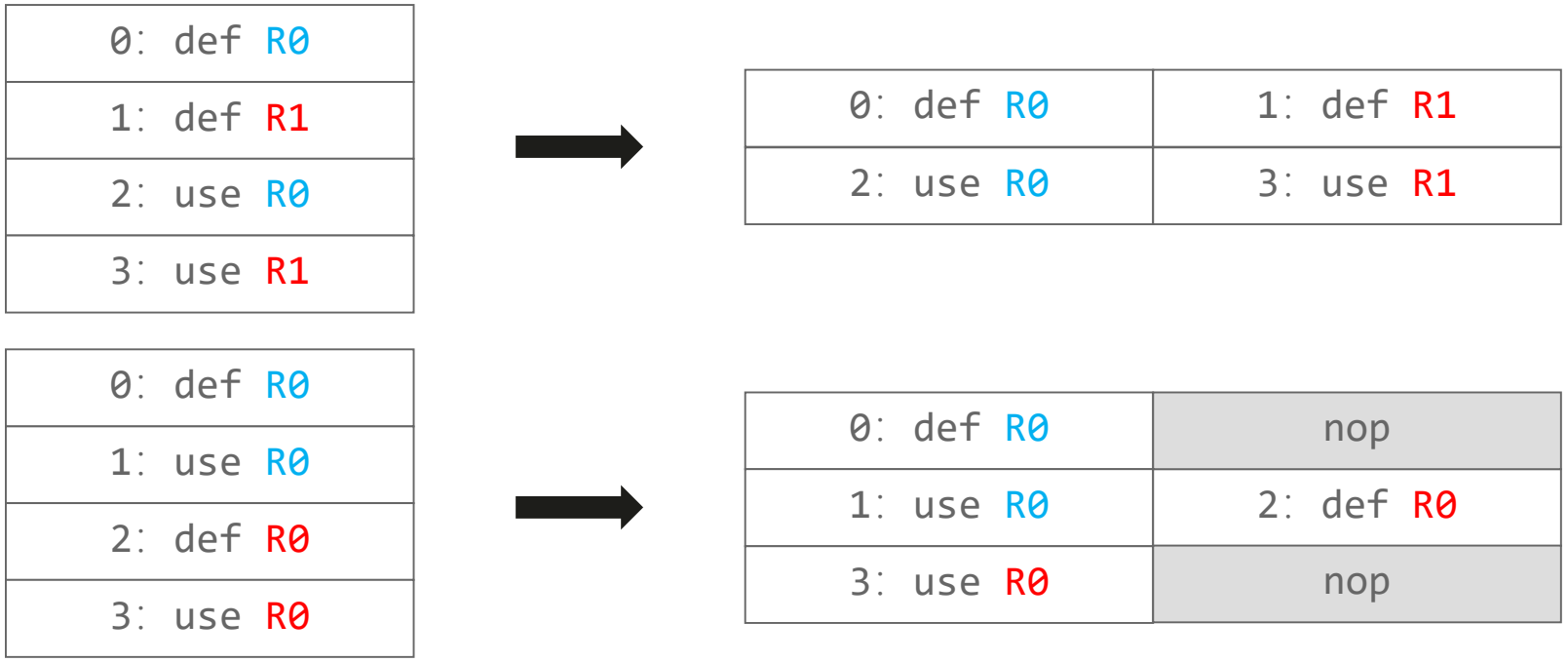
不同的Region适用不同的调度策略，所有Region采用同一策略，很难获得整体最优的性能。改进后的MachineScheduler对于不同Region，自动选择最优策略。更细的粒度，更好的性能。

Post-RA调度与打包

Post-RA调度作用：寄存器分配可能会破坏Pre-RA的调度结果，比如引入spill等， Pre-RA的结果不一定可以提供最优的排布。因此引入Post-RA调度对代码块进行重新排布，为打包Pass提供更好的输入。

打包（Packetizer）：打包是VLIW架构下最重要的Pass之一，将多条无依赖的指令合并成了一个bundle，一个良好的打包相比于不打包，至少提升50%（以PXXA为例，每个bundle平均包含指令~1.5条）。

假设有一个双发射的VLIW处理器：



Pre-RA的更优的调度结果，不一定会为Post-RA和打包带来更优的结果。更好的启发式算法，或者回溯可能能带来整体更优的结果。

Post-RA调度—负边调度

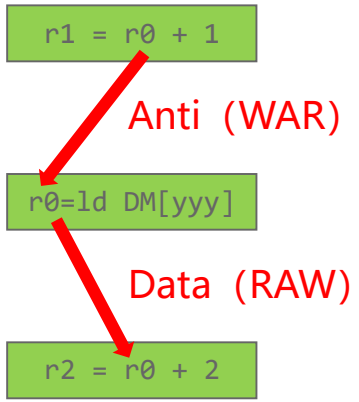
负边调度指的是两条指令的依赖边的延时小于0，常见的有两个场景：

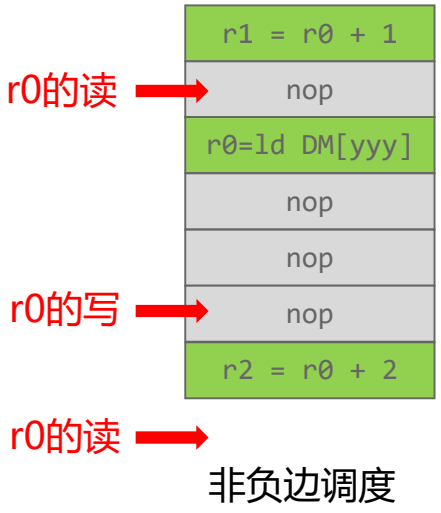
1. delayslots，一些架构（比如PXXV，MIPS）的跳转指令需要在N个cycle后才能真正完成，因此，在跳转指令后有两个cycle，可以用来填充其他指令。
2. 流水线较长的情况下，Succ指令的写行程大于Pred指令的读（或写），那么Succ的指令顺序可以位于Pred之前。

考虑下面的场景，其中：

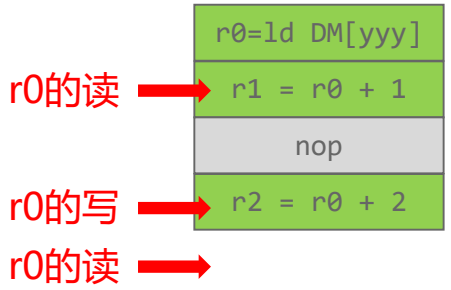
1. 加法的读需要1个cycle，写需要2个cycle
2. ld的写需要3个cycle

单发射的情况下，负边调度相比于非负边调度，cycle由7减少到了4。多发射的情况下，效果会变得更好。



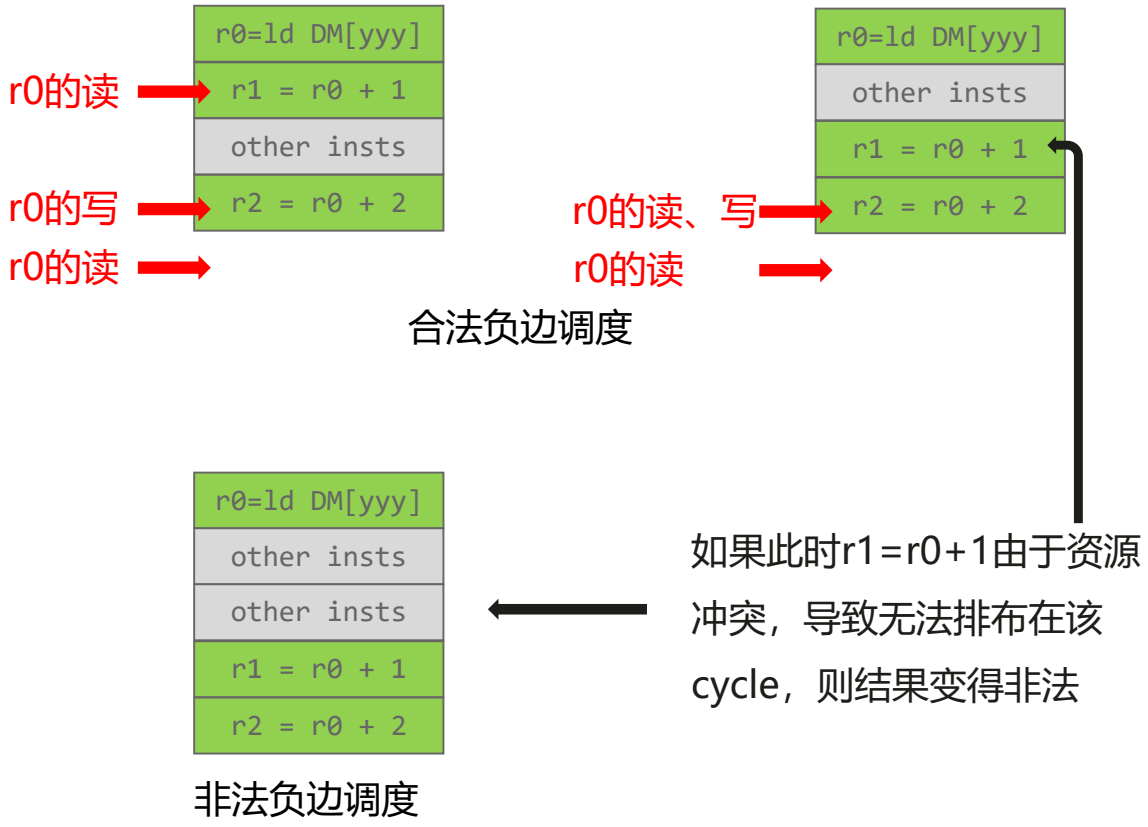


非负边调度



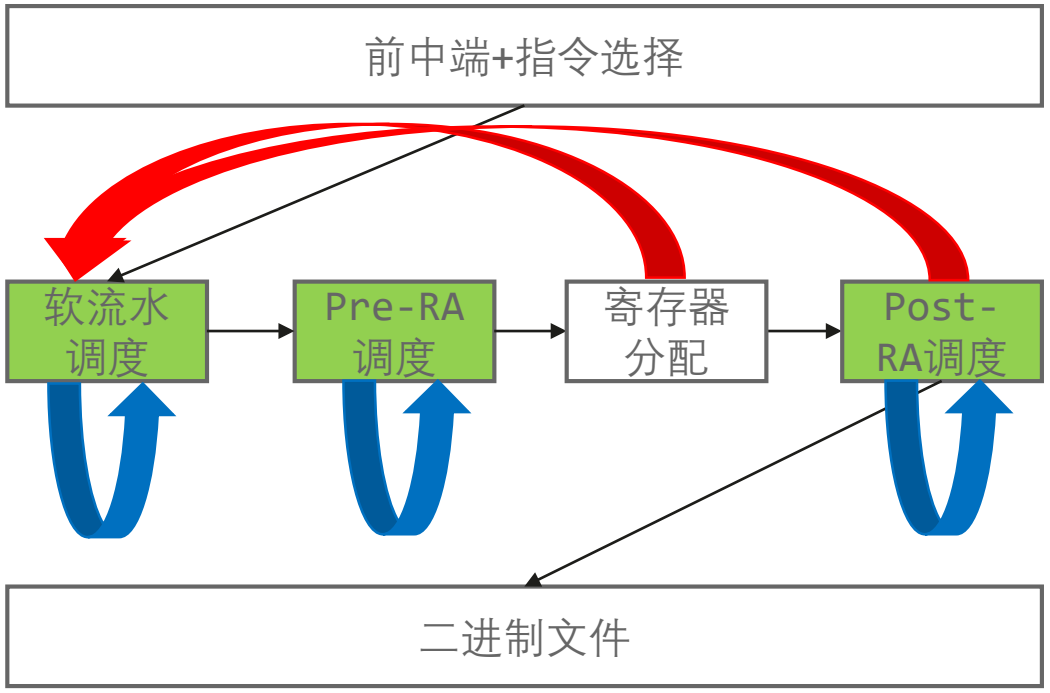
负边调度

Post-RA调度—负边调度失败如何解决



解决方法：回溯重新调度，增大负边的latency的值，比如原先Anti的latency=(1-3)=-2 cycles。现在修改为-1 cycles，可以使有负边依赖的指令更紧密。如果-1依旧不满足，则减少至0，当为0时，其依赖一定满足。

下一步思考：基于Backtracking流程的性能优化



内环

1、软流水内部尝试多种软流水算法，提高软流水成功率和效率；
2、Pre-RA提供top-down, bottom-up, bidirectional，non-sched四种调度方式，适配不同类型的BasicBlock
3、 Post-RA在top-down调度方式的基础上，提供部分指令回退并重新调度的能力，避免非法情况的出现，并尽可能提高IPC

中环

目标提供任意Pass之间的回路，以编译时间换优化空间。目前提供RA->软流水及Post-RA->软流水的回路，对循环展开不同次数，尝试扩大寻优空间，从各种合法结果中择优。

外环

PGO，利用动态仿真信息，调整编译策略，提高程序执行效率。

Thank you.

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

Bring digital to every person, home and organization for a fully connected, intelligent world.

HUAWEI