9 Instruction Scheduling

Chapter Contents

Instruction Scheduling

On most modern processors, the order in which instructions appear has an impact on the speed with which the code executes.

Processors overlap the execution of operations, issuing successive operations as quickly as possible given the finite (and small) set of functional units.

The difficulty arises when an operation issues before its operands are ready

- the processor can stall the premature operation until its operands are available
- the processor can execute the premature operation, albeit with the incorrect operands

Note The second approach relies on the scheduler to maintain enough distance between a value's definition and its various uses to maintain correctness.

Instruction Scheduling

Terminology

A **stall** is the delay caused by a hardware interlock that prevents a value from being read until its defining operation completes.

An **interlock** is the mechanism that detects the premature issue and creates the actual delay.

A processor that relies on compiler insertion of `nop`s for correctness is a **statically scheduled** processor.

A processor that provides interlocks to ensure correctness is a **dynamically scheduled** processor.

Instruction Scheduling

Commodity microprocessors often have operations that have **different latencies**. The following table summarizes typical values.

Cycles	Operation
1	integer add or subtract
3	integer multiply or a floating-point add or subtract
5	floating-point multiply
12–18	floating-point divide
20–40	integer divide

The latency of a load operation can range from a few cycles, *e.g.*, 1–5 cycles for the nearest cache, to tens or hundreds of cycles for values in main memory.

Instruction Scheduling

Many commodity processors have the property that they can initiate execution of **more than one** operation in each cycle.

- **superscalar** processors exploit parallelism at the instruction level, *i.e.*, independent operations that can run concurrently without conflict
- **very long instruction word** (VLIW) processors issue an operation for each functional unit in each cycle, all gathered into a single fixed-format instruction

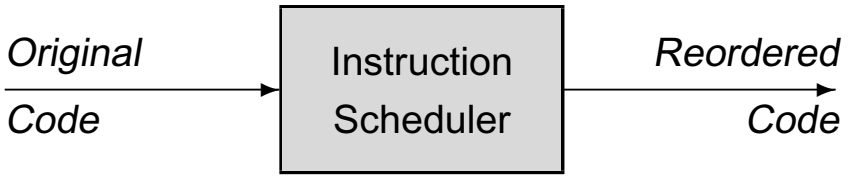
While the mechanisms vary across architectures, the underlying challenge for the scheduler is the same: make good utilization of the hardware resources.

This diversity of hardware dispatch mechanisms blurs the distinction between an operation and an instruction. We will use these terms as follows.

- **operation**: a single opcode and its operands
- **instruction**: aggregation of one or more operations that all issue in the same cycle

Instruction Scheduling

Informally, instruction scheduling[3] is the process whereby a compiler reorders the operations in the compiled code in an attempt to decrease its running time.



The scheduler assumes a fixed set of operations: it **does not rewrite** the code (other than adding nops to maintain correct execution).

The scheduler assumes a fixed allocation of values to registers: while it may rename registers, it **does not change** allocation decisions.

[3] In deference to tradition, we still refer to this problem as *instruction* scheduling, although it might be more precisely called *operation* scheduling

Instruction Scheduling

The instruction scheduler has **three** primary goals

- it must preserve the meaning of the code that it receives as input
- it should minimize execution time by avoiding stalls or nops
- it should avoid increasing value lifetimes past the point where additional register spills are necessary

Of course, the scheduler should also operate **efficiently**.

We will introduce the **list-scheduling algorithm** for a single-issue machine and point out how to extend the basic algorithm to handle multi-operation instructions.

Problem Statement

Assume that the processor has a single functional unit, loads and stores take three cycles, a multiply takes two cycles, and all other operations complete in a single cycle.

Start	End				
1	3	loadAI	r_{arp}, @v1	⇒	r_1
4	4	add	r_1, r_1	⇒	r_1
5	7	loadAI	r_{arp}, @v2	⇒	r_2
8	9	mult	r_1, r_2	⇒	r_1
10	12	loadAI	r_{arp}, @v3	⇒	r_2
13	14	mult	r_1, r_2	⇒	r_1
15	17	loadAI	r_{arp}, @v4	⇒	r_2
18	19	mult	r_1, r_2	⇒	r_1
20	22	storeAI	r_1	⇒	r_{arp}, @a

Start	End				
1	3	loadAI	r_{arp}, @v1	⇒	r_1
2	4	loadAI	r_{arp}, @v2	⇒	r_2
3	5	loadAI	r_{arp}, @v3	⇒	r_3
4	4	add	r_1, r_1	⇒	r_1
5	6	mult	r_1, r_2	⇒	r_1
6	8	loadAI	r_{arp}, @v4	⇒	r_2
7	8	mult	r_1, r_3	⇒	r_1
9	10	mult	r_1, r_2	⇒	r_1
11	13	storeAI	r_1	⇒	r_{arp}, @a

The original code, shown on the left, takes 22 cycles. The scheduled code, shown in the right, reduces the running time to 13 cycles, a 41% improvement.

Problem Statement

The instruction scheduling problem is defined over the **dependence graph** $\mathcal{D}$ of a basic block.

- edges in $\mathcal{D}$ represent the flow of values in the block
- each node has two attributes, an operation *type* and a *delay*

For a node n, the operation corresponding to n must execute on a functional unit specified by its operation *type*. It requires *delay*(n) cycles to complete.

Dependence Graph

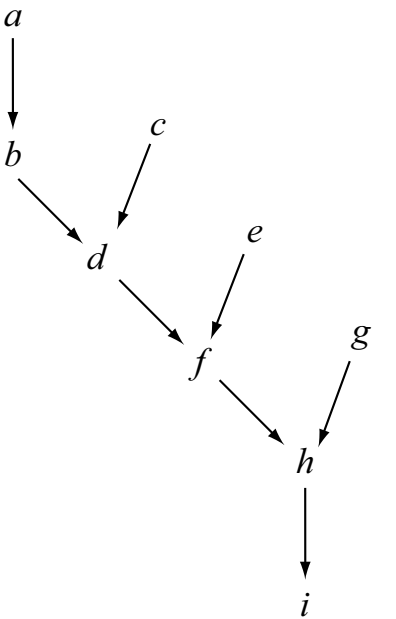
For a block b, its dependence graph $\mathcal{D} = (N, E)$ has a node for each operation in b. An edge in $\mathcal{D}$ connects two nodes n_1 and n_2 if n_2 uses the result of n_1.

Problem Statement

Example The dependence graph for the code below is shown on the right.

Node

```
a   loadAI   r_arp, @v1 ⇒ r1
b   add      r1, r1     ⇒ r1
c   loadAI   r_arp, @v2 ⇒ r2
d   mult     r1, r2     ⇒ r1
e   loadAI   r_arp, @v3 ⇒ r2
f   mult     r1, r2     ⇒ r1
g   loadAI   r_arp, @v4 ⇒ r2
h   mult     r1, r2     ⇒ r1
i   storeAI  r1         ⇒ r_arp, @a
```



Note In general, $\mathcal{D}$ is not a tree. Rather, it is a forest of DAGs. Thus, nodes can have multiple parents and $\mathcal{D}$ can have multiple roots.

Problem Statement

Given a dependence graph $\mathcal{D}$ for a code fragment, a schedule S maps each node $n \in N$ to a non-negative integer that denotes the cycle in which it should be issued, assuming that the first operation issues in cycle 1.

This provides a clear and concise definition of an instruction, namely, the i^{th} instruction is the set of operations $\{ n \mid S(n) = i \}$.

A schedule must meet **three** constraints

- $\forall n \in N : S(n) \geq 1$: operations that issue before execution starts are forbidden
- $\forall (n_1, n_2) \in E : S(n_1) + \textit{delay}(n_1) \leq S(n_2)$: an operation cannot issue until its operands have been defined
- each instruction contains no more operations of each type t than the target machine can issue in a cycle

Problem Statement

Given a well-formed schedule that is both correct and feasible, the **length** of the schedule is the cycle number in which the last operation completes, assuming the first instruction issues in cycle 1. Schedule length can be computed as follows.

$$L(S) = \max_{n \in N}(S(n) + \mathit{delay}(n))$$

With a notion of schedule length comes the notion of a **time-optimal schedule**.

Time-optimal Schedule

A schedule S_i is time optimal if $L(S_i) \leq L(S_j)$ for all other schedules S_j that contain the same set of operations.

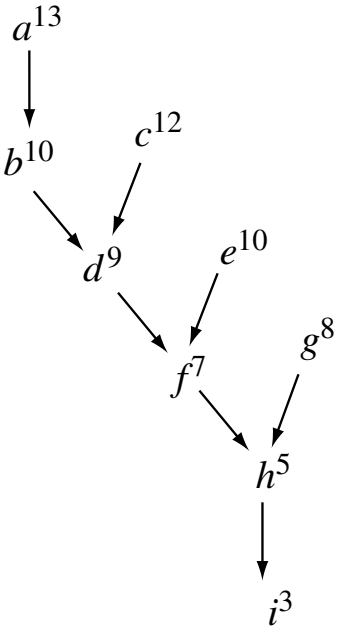
Problem Statement

Computing the total delay along the paths through the graph exposes additional detail about the block.

Annotating the dependence graph $\mathcal{D}$ for our example with information about cumulative latency yields the graph shown on the right.

The path length from a node to the end of the computation is shown as a superscript on the node.

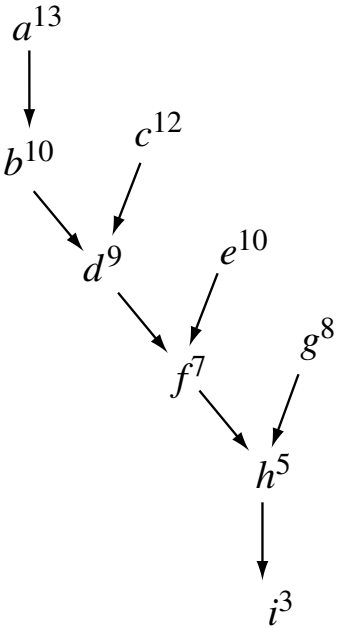
The values clearly show that the path `abdfhi` is longest. It is the **critical path** that determines overall execution time for this example.



Problem Statement

Example Using cumulative latency information for scheduling.

- since $\mathtt{a}$, $\mathtt{c}$, e, and $\mathtt{g}$ have no predecessors in the graph, they are the initial candidates for scheduling
- the fact that $\mathtt{a}$ lies on the critical path strongly suggests that it be scheduled into the first instruction
- once $\mathtt{a}$ has been scheduled, the longest path remaining in $\mathcal{D}$ is $\mathtt{cdefhi}$, suggesting that $\mathtt{c}$ be scheduled as the second instruction
- with the schedule $\mathtt{ac}$, $\mathtt{b}$ and e tie for the longest path
- however, $\mathtt{b}$ needs the result of $\mathtt{a}$, which will not be available until the fourth cycle, making e followed by $\mathtt{b}$ the better choice
- continuing in this fashion leads to the schedule $\mathtt{acebdgfhi}$



Note The resulting schedule matches the schedule shown on the right of Slide 478.

Problem Statement

The compiler cannot simply rearrange the instructions into the proposed order $acebdgfhi$. Both c and e define r_2 and d uses the value that c stores in r_2.

```
c  loadAI  r_arp, @v2  ⇒  r_2
d  mult    r_1, r_2    ⇒  r_1
e  loadAI  r_arp, @v3  ⇒  r_2
```

The scheduler cannot move e before d unless it renames the result of e to avoid the conflict with c's definition of r_2.

This constraint arises not from the flow of data, as with the dependences modelled by edges in $\mathcal{D}$. Instead, it prevents an assignment that would change the flow of data.

These constraints are often called **antidependences**. We denote the antidependence between e and d as $e \rightarrow d$.

Problem Statement

Antidependence

Operation x is antidependent on operation y, denoted as $y \rightarrow x$, if x precedes y and y defines a value used in x.

Reversing their order of execution could cause x to compute a different value.

The scheduler can produce correct code in at least two different ways

- it can discover the antidependences that are present in the input code and **respect** them in the final schedule
- it can systematically **rename** the values in the block to eliminate antidependences before it schedules the code

Local List Scheduling

List scheduling is a **greedy, heuristic approach** to scheduling the operations in a basic block. It has been the dominant paradigm for instruction scheduling since the late 1970s.

- discovers reasonable schedules
- adapts easily to changes in computer architectures

To apply list scheduling to a block, the scheduler follows a **four-step plan**

1. rename to avoid antidependences
2. build a dependence graph $\mathcal{D}$
3. assign priorities to each operation
4. iteratively select an operation and schedule it

The heart of the algorithm, and the key to understanding it, lies in the final step, *i.e.*, the scheduling algorithm.

```
1  Cycle ← 1
2  Ready ← leaves of 𝒟
3  Active ← ∅
4  while Ready ∪ Active ≠ ∅ do
5      foreach op ∈ Active do
6          if S(op) + delay(op) < Cycle then
7              remove op from Active
8              foreach successor s of op in 𝒟 do
9                  if s is ready then
10                     Ready ← Ready ∪ {s}
11     if Ready ≠ ∅ then
12         remove an op from Ready
13         S(op) ← Cycle
14         Active ← Active ∪ {op}
15     Cycle ← Cycle + 1
```

The algorithm maintains a simulation clock `Cycle` to track time.

`Ready` holds all the operations that can execute in the current cycle.

`Active` holds all operations that were issued in an earlier cycle but have not yet finished.

At each time step, it accounts for any operations completed in the previous cycle, it schedules an operation for the current cycle, and it increments `Cycle`.

Local List Scheduling

The **quality** of the schedule produced by this algorithm depends primarily on the mechanism used to pick an operation from the `Ready` queue.

- the algorithm should take the operation with the **highest priority score**
- in the case of a tie, it should use **one or more other criteria** to break the tie

The metric suggested earlier, *i.e.*, the longest latency-weighted distance to a root in $\mathcal{D}$, corresponds to always choosing the node on the **critical path** for the current cycle.

To the extent that the impact of a scheduling priority is predictable, this scheme should provide **balanced** pursuit of the longest paths.

Scheduling Operations with Variable Delays

Memory operations often have **uncertain** and **variable** delays.

- assuming the **worst-case** delay, risks idling the processor for long periods
- assuming the **best-case** delay, will stall the processor on a cache miss

The compiler can obtain good results by calculating an individual latency for each load based on the amount of instruction-level parallelism available to cover the load's latency.

This approach, called **balanced scheduling**, schedules the load with regard to the code that surrounds it rather than the hardware on which it will execute.

- mitigates cache misses by scheduling as much extra delay as possible for each load
- will not slow down execution in the absence of cache misses

Scheduling Operations with Variable Delays

```
1 foreach load operation l in the block do
2     delay(l) ← 1
3 foreach operation op in 𝒟 do
4     let 𝒟_op be the nodes and edges in 𝒟 independent of op
5     foreach connected component 𝒞 of 𝒟_op do
6         find the maximal number of loads N on any path through 𝒞
7         foreach load operation l in 𝒞 do
8             delay(l) ← delay(l) + delay(op)/N
```

Using this value as $\mathit{delay}(\mathrm{l})$ produces a schedule that **shares the slack time** of independent operations evenly across all loads in the block.

Extending the Algorithm

The list-scheduling algorithm, as presented, makes several **assumptions** that may not hold true in practice.

1. most processors can issue **multiple operations per cycle**: scheduler needs to look for an operation for each functional unit in each cycle
2. some operations can execute on **multiple** functional units and others cannot: scheduler must chose an order for functional units
3. **partitioned** register set: scheduler may need to place an operation in the partition where its operands reside
4. **block boundaries**: invoke scheduler on blocks in reverse postorder on the CFG to understand when operands computed in predecessor blocks are available

Tie Breaking in the List-Scheduling Algorithm

In practice, list scheduling produces good results: it often builds optimal or near-optimal schedules. However, as with many greedy algorithms, its behavior is **not robust**.

The methodology used to **break ties** has a strong impact on the quality of schedules.

A typical scheduler has two or three tie-breaking **priority ranks** for each operation, which it applies in a consistent order.

- **node's latency-weighted path length**
- **number of node's immediate successors**: encourages the scheduler to pursue many distinct paths through the graph
- **total number of node's descendants**: amplifies the effect of the previous ranking
- **node's delay**: schedules long-latency operations as soon as possible
- **number of operands for which node is the last use**: moves last uses closer to their definitions

Forward versus Backward List Scheduling

As presented, the list-scheduling algorithm works over the dependence graph **from its leaves to its roots** and creates the schedule from the first cycle in the block to the last.

An alternate formulation of the algorithm works over the dependence graph in the **opposite direction**, scheduling from roots to leaves.

This version of the algorithm is called **backward list scheduling**, and the original version is called **forward list scheduling**.

List scheduling is not an expensive part of compilation. Some compilers run the scheduler **several times** with different combinations of heuristics and keep the best schedule.

Forward versus Backward List Scheduling

In practice, neither forward scheduling nor backward scheduling always wins.

Critical operations occur near the leaves

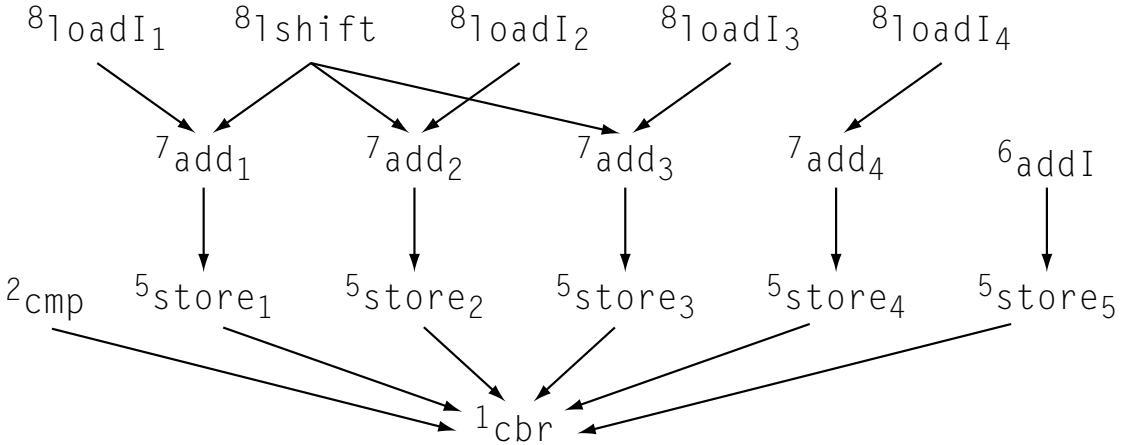
- forward scheduling seems more likely to consider them together
- backward scheduling must work its way through the remainder of the block to reach them

Critical operations occur near the roots

- backward scheduling may examine them together
- forward scheduling sees them in an order dictated by decisions made starting at the other end of the block

Forward versus Backward List Scheduling

Example Below the dependence graph for a basic block found in the SPEC 95 benchmark program `go` is shown.



Opcode	Latency
`loadI`	1
`lshift`	1
`add`	2
`addI`	1
`cmp`	1
`store`	4

Note The compiler added dependences from the `store` operations to the block-ending branch to ensure that the memory operations complete before the next block begins.

	Integer	Integer	Memory
1	$loadI_1$	lshift	–
2	$loadI_2$	$loadI_3$	–
3	$loadI_4$	add_1	–
4	add_2	add_3	–
5	add_4	addI	$store_1$
6	cmp	–	$store_2$
7	–	–	$store_3$
8	–	–	$store_4$
9	–	–	$store_5$
10	–	–	–
11	–	–	–
12	–	–	–
13	cbr	–	–

	Integer	Integer	Memory
1	$loadI_4$	–	–
2	addI	lshift	–
3	add_4	$loadI_3$	–
4	add_3	$loadI_2$	$store_5$
5	add_2	$loadI_1$	$store_4$
6	add_1	–	$store_3$
7	–	–	$store_2$
8	–	–	$store_1$
9	–	–	–
10	–	–	–
11	cmp	–	–
12	cbr	–	–
13	–	–	–

Forward versus Backward List Scheduling

The five `store` operations take most of the time in the block. The schedule that minimizes execution time must begin executing stores **as early as possible**.

The backward schedule places the `addI` earlier in the block, allowing $\texttt{store}_5$ to issue in cycle 4, *i.e.*, **one cycle earlier** than the first memory operation in the forward schedule.

Note By considering the problem in a **different** order, using the **same** underlying priorities and tie breakers, the backward algorithm finds a different result.

Regional List Scheduling

Moving from single basic blocks to **larger scopes** can improve the quality of code that the compiler generates.

For instruction scheduling, many different approaches have been proposed for regions **larger than a block** but **smaller than a whole procedure**.

Almost all of those approaches use basic list-scheduling and **surround** that algorithm with an infrastructure that lets it consider longer (*e.g.*, multi-block) sequences of code.

- extended basic blocks
- traces
- cloning

Scheduling Extended Basic Blocks

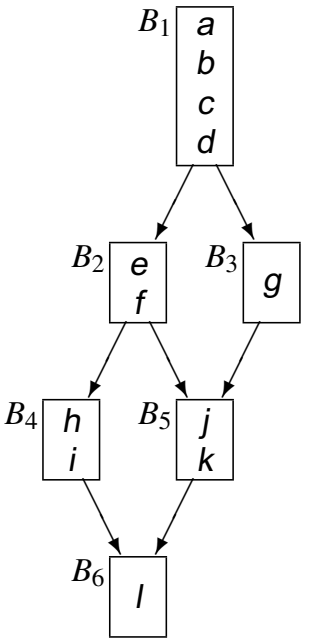
An **extended basic block** (EBB) consists of a set of blocks $B_1, B_2, \ldots, B_n$ in which B_1 has multiple predecessors and every other block B_i has exactly one predecessor, some B_j in the EBB.

Example The CFG on the right has one large EBB, $\{B_1, B_2, B_3, B_4\}$, and two trivial EBBs, $\{B_5\}$ and $\{B_6\}$.

To obtain a **larger context** for list scheduling, the compiler can treat paths in an EBB, such $\{B_1, B_2, B_4\}$, as if they were single blocks.

When scheduling paths, the compiler needs to accounts for

- shared path prefixes, *e.g.*, B_1, which occurs in $\{B_1, B_2, B_4\}$ and $\{B_1, B_3\}$
- premature exits, *e.g.*, $B_1 \rightarrow B_3$ and $B_2 \rightarrow B_5$



Scheduling Extended Basic Blocks

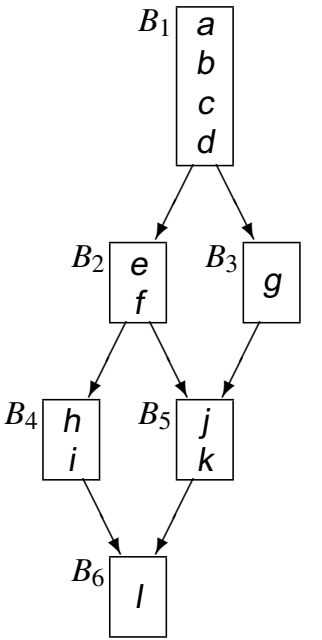
To see how **shared prefixes** and **premature exits** complicate list scheduling, consider the possibilities for code motion in $\{B_1, B_2, B_4\}$.

Move an operation **forward**, *e.g.*, c from B_1 to B_2

- might speed execution along the path $\{B_1, B_2, B_4\}$
- changes the computation performed along the path $\{B_1, B_3\}$
- to fix this problem, the scheduler must insert a copy of c into B_3

Move an operation **backward**, *e.g.*, f from B_2 to B_1

- might speed execution along the path $\{B_1, B_2, B_4\}$
- inserts a computation of f into the path $\{B_1, B_3\}$, which might produce incorrect code along this path
- the scheduler must rewrite the code to undo that effect in B_3



Scheduling Extended Basic Blocks

Compensation Code

Code inserted into a block B_i to **counteract** the effects of cross-block code motion along a path that does not include B_i.

The issue of compensation code also makes clear the **order** in which the scheduler should consider paths in an EBB

→ since the first path scheduled receives little or no compensation code, the scheduler should choose paths in order of their **likely execution frequency**.

Trace Scheduling

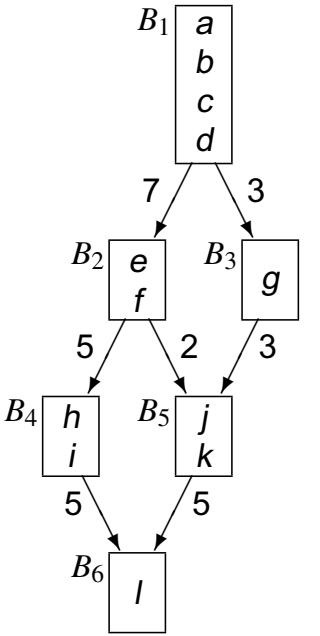
Trace scheduling constructs maximal-length acyclic paths through the CFG and applies the list-scheduling algorithm to those paths, or **traces**.

To build a trace, a simple **greedy approach** can be used that stops when it either runs out of possible edges or encounters a loop-closing branch.

Example The trace for the example on the right is $\{B_1, B_2, B_4, B_6\}$.

With respect to EBB scheduling, one **additional** opportunity for compensation code occurs: a trace may have interim entry points, *i.e.*, blocks in mid-trace that have **multiple** predecessors.

Note EBB scheduling can be considered a degenerate case of trace scheduling in which interim entries to the trace are prohibited.



Cloning for Context

Join points in the CFG **limit the opportunities** for either EBB scheduling or trace scheduling.

The compiler can **clone blocks** to create longer join-free paths.

- for EBB scheduling, it increases the size of the EBB and the length of some of the paths through the EBB
- for trace scheduling, it avoids the complications caused by interim entry points in the trace

After cloning, the entire graph forms **one single** EBB.

The compiler can combine blocks that are linked by an edge where the source has no other successors and the sink has no other predecessors.