Lecture 1:
Instruction Scheduling

CESE4085 Modern Computer Architecture Course
Part 2, L8.1
Carlo Galuzzi

TUDelft

LECTURE CONTENTS

1. Introduction	What is instruction scheduling and why it matters
2. Architectural Background	Pipeline depth, latency, and ISA constraints
3. Scheduling Methodologies	Local, regional, global techniques
4. Software Pipelining	Loop optimization through overlapping
5. Instruction Scheduling Complexity	NP-completeness and heuristic approaches
6. AI and Scheduling	ML-guided and future-oriented methods
7. Advanced Topics	DAGs, IR, register pressure, energy, vector ISAs
8. Case Studies and Applications	Real-world and RISC-V-specific scheduling
9. Summary	Final thoughts and future directions

LEARNING GOALS:

This lecture aims to provide students with an in-depth understanding of instruction scheduling within modern compiler backends, with a focus on LLVM and RISC-V.

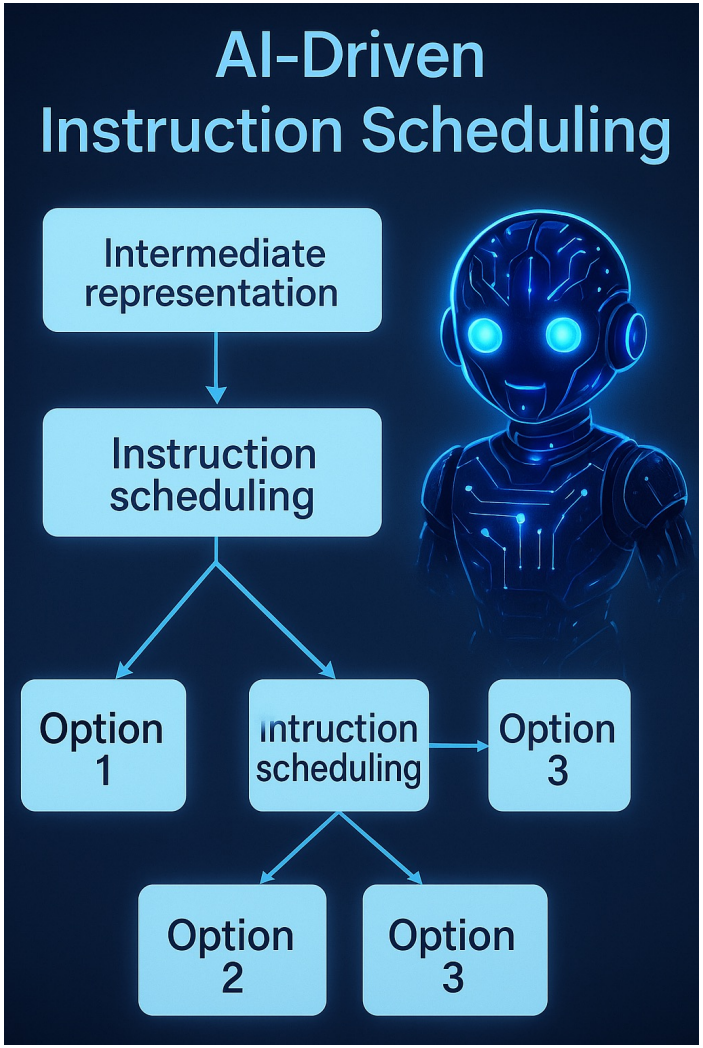
- Understand the role and impact of instruction scheduling on performance.
- Learn the distinctions between local, regional, and global scheduling.
- Analyze architectural factors that influence scheduling decisions.
- Explore software pipelining and its advantages in loop optimization.
- Examine how AI techniques are integrated into modern compilers for scheduling.

EXTRA READING MATERIAL IN BRIGHSPACE

Instruction Scheduling: LLVM and RISC-V Architectures

Table of Contents

1. Introduction
2. Background: Architectural Features That Affect Performance
3. Architectural Features That Affect Performance
4. Overview of Instruction Scheduling Methodologies
5. Background:
    - The Instruction Scheduling Problem
    - Complexity and Heuristic Approaches
6. Local Scheduling: The Algorithm
    - Renaming
    - Building the Dependence Graph
    - Computing Priorities
    - List Scheduling
    - Forward Versus Backward List Scheduling
7. Regional Scheduling
    - Superlocal Scheduling
    - Trace Scheduling
    - Cloning for Context
8. Software Pipelining
    - The Strategy Behind Software Pipelining
    - An Algorithm for Software Pipelining
    - A Final Example
9. AI and Instruction Scheduling
    - Current State of AI in Instruction Scheduling
    - Future Perspectives on AI in Instruction Scheduling
10. Summary and Perspective
11. References
12. Appendix: LLVM Commands for Instruction Selection (RISC-V)



(AI Generated)

INTRODUCTION TO INSTRUCTION SCHEDULING

Instruction scheduling is a critical compiler optimization that rearranges the order of instructions to improve program performance by efficiently utilizing processor resources. Instruction Scheduling:

- **optimizes instruction execution**: by reordering instructions to enhance efficiency.
- **maximizes pipeline throughput**: by ensuring continuous instruction flow without idle cycles.
- **minimizes pipeline stalls**: by reducing waiting caused by instruction dependencies.

Instruction Scheduling is a **Critical compiler optimization**: Essential for achieving peak performance in modern compilers like LLVM targeting RISC-V.

HISTORICAL BACKGROUND

Instruction scheduling techniques first appeared alongside pipeline processor designs in the late 1970s and early 1980s, evolving significantly as processor architectures became more complex.

- **Emergence with pipelines**: Addressed early pipeline stall and efficiency issues.
- **Initial focus on basic blocks**: Simple instruction sequences optimized within isolated blocks.
- **Advanced scheduling techniques**: Trace scheduling and software pipelining developed for handling more complex instruction sequences. (mid-1980s onwards)
- **Modern compiler importance**: Continues as a core component in optimizing compiler outputs.

IMPORTANCE IN MODERN COMPUTING

Instruction scheduling remains highly relevant in modern computing environments due to its significant impact on instruction-level performance and overall system efficiency.

- **Direct performance impa**ct: Directly influences execution speed and program efficiency.
- **Essential for complex processors**: Particularly critical for architectures with deep pipelines and multiple execution units.
- **LLVM's robust capabilities**: Advanced scheduling features specifically optimized for modern architectures like RISC-V.
- **Compiler design relevance**: Plays a pivotal role in modern compiler performance strategies.

ROLE OF INTERMEDIATE REPRESENTATIONS (IR) IN SCHEDULING

Intermediate Representations (IRs) serve as the foundation for scheduling decisions within LLVM.

Purpose:

- Abstract program structure from source code to machine code

Importance:

- Enables target-independent analysis and transformations

Example:

- LLVM's SelectionDAG and MachineInstr levels help manage scheduling-specific information like latency and dependencies

DAG-BASED INSTRUCTION SCHEDULING

Directed Acyclic Graphs (DAGs) are key to representing dependencies and guiding instruction order.

- **Nodes**: Instructions
- **Edges**: Dependencies (data, control)
- **Benefit**:
  - Enables both forward and backward scheduling strategies
- **Example**:
  - LLVM constructs a DAG in the MachineScheduler to plan optimal orderings

ARCHITECTURAL FEATURES INFLUENCING SCHEDULING

Effective instruction scheduling depends significantly on understanding and leveraging hardware architectural characteristics to achieve optimal performance.

- **Pipeline depth**: Influences how instructions must be ordered to avoid stalls.
- **Functional unit availability**: Determines opportunities for parallel instruction execution.
- **Execution latency**: Requires scheduling to consider varying instruction completion times.
- **Branch prediction**: Important for strategically organizing instructions to minimize misprediction penalties.

PIPELINE DEPTH

Pipeline depth greatly affects scheduling strategies by influencing how instructions flow through processor stages. Deeper pipelines can boost throughput but also raise sensitivity to pipeline hazards.

- **Deeper pipelines**: Can execute instructions simultaneously in different stages, improving throughput.
- **Higher stall sensitivity**: Dependent instructions wait longer, causing potential delays.
- **Example**: RISC-V's standard 5-stage pipeline (Fetch, Decode, Execute, Memory, Write-back) demands precise scheduling to avoid pipeline stalls.

EXAMPLE – PIPELINE STALLS

Pipeline stalls occur when dependent instructions cannot execute because earlier instructions have not yet completed.

Example:

```
LOAD x1, 0(x2)
ADD x3, x1, x4
```

- **Explanation**: The ADD instruction depends on the completion of LOAD. The scheduler inserts unrelated instructions between them to keep the pipeline active, avoiding stalls.
- **Importance**: Proper handling of stalls greatly enhances pipeline utilization and processor throughput.

FUNCTIONAL UNIT AVAILABILITY

The **availability of multiple functional units** determines how effectively instructions can be executed in parallel, **directly influencing** scheduling strategies and overall system performance.

- **Multiple functional units** (e.g., ALUs, FPUs): Enable simultaneous execution of independent instructions, maximizing efficiency.
- **Example**: RISC-V processors typically have several arithmetic logic units (ALUs), allowing LLVM to schedule multiple arithmetic operations concurrently.
- **Benefit**: Enhanced parallelism significantly boosts instruction throughput and reduces total execution time.

EXECUTION LATENCY

Instruction latency varies according to instruction types and significantly impacts the efficiency of instruction scheduling decisions.

- **Varied instruction latencies**: Load/store operations usually have longer latencies compared to arithmetic instructions.
- **Scheduler's role**: Must arrange instructions thoughtfully to **mask these latencies**, **optimizing pipeline utilization**.
- **Example**: LLVM scheduler positions independent arithmetic instructions to overlap the longer latency of load operations, thereby minimizing idle cycles.

BRANCH PREDICTION

Branch prediction mechanisms are crucial for reducing pipeline disruptions caused by conditional branches, influencing how schedulers organize instruction sequences.

- **Accurate branch prediction**: Minimizes misprediction penalties, maintaining pipeline efficiency.
- **Scheduling implication**: Instructions following branches are scheduled to optimize execution paths predicted to be taken.
- **Example**: LLVM prioritizes instructions from likely execution paths immediately after conditional branches, reducing stall penalties associated with incorrect predictions.

CACHE HIERARCHY & REGISTER FILE

Efficient instruction scheduling must consider the processor's cache hierarchy and register availability to optimize memory access and reduce register spills.

- **Cache hierarchy:**
  - Scheduling memory accesses sequentially can improve cache hit rates.
  - Reducing cache misses significantly decreases latency.
- **Register file:**
  - Optimizing register reuse avoids frequent load/store operations.
  - Reducing register spills directly enhances execution speed.
- **Example**:
  - LLVM schedules related instructions closely to exploit cache locality and efficient register usage.

OUT-OF-ORDER EXECUTION

Schedulers can optimize instruction arrangements to support **out-of-order execution**, **where hardware dynamically reorders instructions to enhance pipeline utilization**.

- **Out-of-order execution benefits:**
  - Reduces stalls by executing independent instructions first.
  - Maximizes resource utilization and throughput.
- **Scheduler's role:**
  - Reduces dependencies that constrain instruction ordering.
  - Facilitates hardware's dynamic scheduling capabilities.
- **Example**:
  - LLVM arranges instructions to minimize dependencies, enabling efficient execution in processors capable of out-of-order processing.

INSTRUCTION SCHEDULING METHODOLOGIES OVERVIEW

Instruction scheduling methodologies include **local, regional, and global scheduling**. Each approach addresses **different scopes** and **complexities** to optimize instruction execution.

- **Local scheduling**: Focuses on single basic blocks.
- **Regional scheduling**: Optimizes multiple blocks and paths collectively.
- **Global scheduling**: Manages instructions across entire functions or larger code sections.
- **Example**:
  - LLVM employs these methodologies selectively based on compilation goals and complexity trade-offs.

LOCAL SCHEDULING OVERVIEW

Local scheduling deals with instruction ordering within **individual basic blocks**, aiming for quick compilation and efficient execution.

- **Scope**:
  - Single basic blocks (no internal branches).
- **Characteristics**:
  - Fast, minimal computational overhead.
  - Suitable for JIT (Just-In-Time) compilation scenarios.
- **Example**:
  - LLVM quickly schedules instructions within basic blocks to reduce pipeline stalls and optimize simple sequences.

LOCAL SCHEDULING TECHNIQUES

Several key techniques facilitate **effective local scheduling**, including **renaming**, **constructing dependence graphs**, and **priority calculation**.

- **Renaming**: Removes false dependencies to increase instruction parallelism.
- **Dependence graphs**: Clearly visualize instruction dependencies.
- **Priority calculation**: Determines optimal instruction execution order.
- **List scheduling**: A heuristic that orders instructions by priority for efficient scheduling.

LOCAL SCHEDULING EXAMPLE

Scheduling a basic block with dependent and independent instructions.

```
1: LOAD x1, 0(x2)
2: ADD x3, x1, x4
3: SUB x5, x6, x7
4: MUL x8, x3, x9
```

Analysis:

- 1→2→4: dependency chain.
- 3 is independent.

Optimized Schedule:

```
1: LOAD x1, 0(x2)
2: SUB x5, x6, x7 ; overlaps with load latency
3: ADD x3, x1, x4
4: MUL x8, x3, x9
```

Result: Reduced stalls, improved throughput.

RENAMING EXAMPLE

Renaming involves assigning new registers to eliminate false dependencies, allowing instructions to execute in parallel.

- **False dependency example:**

```
ADD x1, x2, x3
SUB x1, x4, x5
```

- **Explanation**:
  - Both instructions initially reuse x1, causing unnecessary dependency.
  - Renaming assigns distinct registers, enabling simultaneous execution.
- **Result**:

```
ADD x6, x2, x3
SUB x7, x4, x5
```

DEPENDENCE GRAPH CONSTRUCTION

Dependence graphs visually represent instruction dependencies, significantly aiding the scheduling process.

- **Nodes**: Represent individual instructions.
- **Edges**: Indicate data or resource dependencies.
- **Purpose**: Clearly illustrates scheduling constraints and opportunities.
- **Example**:
  - Helps LLVM determine the optimal scheduling order by visualizing dependencies explicitly.

DEPENDENCE GRAPH CONSTRUCTION: EXAMPLE (1)

Consider the following sequence of instructions:

```
1: LOAD x1, 0(x2)
2: ADD x3, x1, x4
3: MUL x5, x3, x6
4: STORE x5, 0(x7)
```

Dependence Graph Representation:

Nodes: Each node represents an instruction:

- `Node 1: LOAD x1, 0(x2)`
- `Node 2: ADD x3, x1, x4`
- `Node 3: MUL x5, x3, x6`
- `Node 4: STORE x5, 0(x7)`

DEPENDENCE GRAPH CONSTRUCTION: EXAMPLE (2)

Consider the following sequence of instructions:

```
1: LOAD x1, 0(x2)
2: ADD x3, x1, x4
3: MUL x5, x3, x6
4: STORE x5, 0(x7)
```

Dependence Graph Representation:

Edges: Directed edges represent dependencies between instructions:

- Edge from Node 1 to Node 2 (Node 2 depends on Node 1 for `x1`).
- Edge from Node 2 to Node 3 (Node 3 depends on Node 2 for `x3`).
- Edge from Node 3 to Node 4 (Node 4 depends on Node 3 for `x5`).

DEPENDENCE GRAPH CONSTRUCTION: EXAMPLE (3)

Consider the following sequence of instructions:

```
1: LOAD x1, 0(x2)
2: ADD x3, x1, x4
3: MUL x5, x3, x6
4: STORE x5, 0(x7)
```

Dependence Graph Representation:

Visual Representation:

```
[LOAD x1] --> [ADD x3] --> [MUL x5] --> [STORE x5]
```

This clear graphical representation helps the scheduler easily identify instruction dependencies and make informed scheduling decisions to optimize pipeline utilization and minimize execution stalls.

ANALYSIS OF INSTRUCTION DEPENDENCIES

Understanding dependency types is key for scheduling correctness and efficiency.

Types:

- Read After Write (RAW): true dependency
- Write After Read (WAR): anti-dependency
- Write After Write (WAW): output dependency

Handling:

- Renaming for WAR/WAW
- Dependency graphs for RAW

Example:

```
1: ADD x1, x2, x3
2: SUB x2, x4, x5 ; WAR
3: MUL x6, x1, x7 ; RAW
```

Scheduler resolves 2 with renaming and respects 3's order

PRIORITY CALCULATION

Priority calculation determines the **optimal order of instruction** scheduling based on multiple criteria.

- **Factors influencing priority:**
  - Instruction latency
  - Resource availability
  - Dependency chains

- **Importance**:
  - Higher-priority instructions are scheduled first to optimize pipeline efficiency.

- **Example**:
  - High-latency load instructions are prioritized to overlap their execution with other independent instructions.

PRIORITY CALCULATION: EXAMPLE (1)

Consider the following instructions with their associated latencies:

Instruction	Latency (cycles)	Dependencies
LOAD x1	3	None
ADD x2	1	LOAD x1
MUL x3	2	ADD x2
SUB x4	1	LOAD x1
DIV x5	4	SUB x4, MUL x3

Priority Calculation Approach:

1. **Identify dependencies:**
    1. ADD x2 depends on LOAD x1.
    2. MUL x3 depends on ADD x2.
    3. SUB x4 depends on LOAD x1.
    4. DIV x5 depends on SUB x4 and MUL x3.

PRIORITY CALCULATION: EXAMPLE (2)

Consider the following instructions with their associated latencies:

Instruction	Latency (cycles)	Dependencies
LOAD x1	3	None
ADD x2	1	LOAD x1
MUL x3	2	ADD x2
SUB x4	1	LOAD x1
DIV x5	4	SUB x4, MUL x3

Calculate priority based on latency and dependency chain:

LOAD x1: Highest priority (latency 3 cycles, foundational dependency).

ADD x2: High priority due to direct dependence on **LOAD x1**.

SUB x4: High priority, independent of **ADD** and **MUL**, depends only on **LOAD x1**.

MUL x3: Medium priority, depends on **ADD x2**.

DIV x5: Lowest priority due to multiple dependencies and longest latency.

PRIORITY CALCULATION: EXAMPLE (3)

Consider the following instructions with their associated latencies:

Instruction	Latency (cycles)	Dependencies
LOAD x1	3	None
ADD x2	1	LOAD x1
MUL x3	2	ADD x2
SUB x4	1	LOAD x1
DIV x5	4	SUB x4, MUL x3

Priority Order:

1. `LOAD x1`
2. `ADD x2, SUB x4` (parallel if possible)
3. `MUL x3`
4. `DIV x5`

This calculated priority helps the scheduler **efficiently organize** the instructions, **optimizing pipeline** utilization by scheduling instructions with higher latency and critical dependencies earlier.

LIST SCHEDULING

List scheduling is a **heuristic** approach that orders instructions based on calculated priorities to optimize instruction execution.

- **Operation**:
  - Maintains a prioritized list of ready-to-execute instructions.
  - Scheduler picks the highest-priority instruction for execution.
- **Advantage**:
  - Simple and computationally efficient.
- **Example**:
  - LLVM schedules load operations ahead of arithmetic operations to overlap latencies effectively.

FORWARD VS. BACKWARD LIST SCHEDULING

Forward and backward list scheduling represent two directional approaches for scheduling instructions.

- **Forward scheduling:**
  - Starts from the beginning of a basic block and schedules forward.
  - Simpler implementation, suitable for most scenarios.
- **Backward scheduling:**
  - Begins scheduling from the end of the block, moving backward.
  - Optimizes resource usage, especially beneficial when resources become available later in execution.
- **Example**:
  - LLVM may employ backward scheduling to optimize registers and resources in tight execution contexts.

REGIONAL SCHEDULING OVERVIEW

Regional scheduling spans multiple basic blocks to achieve more comprehensive optimization than local scheduling.

- **Scope**:
  - Includes inter-block dependencies and execution paths.
- **Purpose**:
  - Optimize regions collectively rather than in isolation.
- **Benefit**:
  - Higher performance gains by considering broader instruction contexts.
- **Example**:
  - LLVM selects code regions based on profiling data to apply regional scheduling and improve execution paths.

REGIONAL SCHEDULING EXAMPLE (1)

Example of optimizing a hot execution path across two blocks.

Block A:

```
1: LOAD x1, 0(x2)
2: BRANCH to B
```

Block B:

```
3: ADD x3, x1, x4
4: MUL x5, x3, x6
```

Regional scheduling across A and B:

- Moves ADD/MUL close to LOAD for tighter dependency resolution.
- Considers branch prediction likelihood.

REGIONAL SCHEDULING EXAMPLE (2)

Example of optimizing a hot execution path across two blocks.

Block A:

```
1: LOAD x1, 0(x2)
2: BRANCH to B
```

Block B:

```
3: ADD x3, x1, x4
4: MUL x5, x3, x6
```

Transformed execution:

```
1: LOAD x1, 0(x2)
2: ADD x3, x1, x4
3: MUL x5, x3, x6
4: BRANCH to B
```

Benefit: Reduced latency between dependent instructions.

SUPERLOCAL SCHEDULING

Superlocal scheduling optimizes instruction execution **across a group of basic blocks** called a **superblock**, which has one entry point but may contain multiple exit points.

- **Concept**:
  - Treats a set of connected basic blocks as one scheduling unit.
- **Purpose**:
  - Focuses optimization on frequently executed control paths.
- **Benefit**:
  - Improves performance by optimizing larger code regions with reduced control flow interruptions.
- **Example**:
  - LLVM constructs superblocks using profiling data and schedules instructions across them for better throughput.

TRACE SCHEDULING

Trace scheduling identifies and optimizes **linear paths** through **multiple basic blocks**, particularly those that are frequently executed (hot paths).

- **Concept**:
  - Linear trace of basic blocks is selected and treated as a unit.
- **Optimizations**:
  - Schedule for the trace, then insert compensation code for off-trace paths.
- **Benefit**:
  - High performance for common execution scenarios.
- **Example**:
  - LLVM reorders instructions along the most probable path in a conditional branch sequence.

CROSS-BASIC-BLOCK SCHEDULING CONSTRAINTS

Regional and global scheduling must navigate control flow and semantic preservation.

Key Constraint:

- Must ensure correctness across conditional branches and loops.

Strategy:

- Use compensation code for out-of-trace execution paths.
- Weight edges with execution likelihood.

LLVM's Approach:

- Relies on control flow graphs and profiling to guide inter-block scheduling, preserving correctness while maximizing gain.

CLONING FOR CONTEXT

Cloning for context creates multiple specialized copies of basic blocks to allow better optimization in different execution contexts.

- **Concept**:
  - Duplicate blocks tailored to specific calling or control-flow paths.
- **Purpose**:
  - Reduce constraint from conflicting contexts.
- **Benefit**:
  - Enables more aggressive and effective scheduling.
- **Example**:
  - LLVM may clone a block executed in both tight loops and conditionals to allow loop-specific scheduling.

SOFTWARE PIPELINING

Software pipelining is a scheduling technique that overlaps instructions from different iterations of a loop to maximize parallelism.

- **Goal**:
  - Initiate new loop iterations before previous ones complete.
- **Benefit**:
  - Increases throughput and better utilizes resources.
- **Key Feature:**
  - Especially effective in loops with independent operations across iterations.
- **Example**:
  - LLVM schedules loop loads, computations, and stores from different iterations in parallel.

SOFTWARE PIPELINING STRATEGY

The strategy of software pipelining is **to break down loops** into **prologue**, **kernel**, and **epilogue** to allow overlapping execution.

- **Phases**:
  - **Prologue**: Sets up initial overlapping.
  - **Kernel**: Steady-state overlapping of loop iterations.
  - **Epilogue**: Completes remaining operations after final iteration.
- **Scheduler's Role:**
  - Assigns loop instructions to available slots while satisfying dependencies.
- **Example:**
  - For loop processing an array, overlapping LOAD of i+1 with compute/store of i improves efficiency.

SOFTWARE PIPELINING ALGORITHM

Software pipelining relies on a **three-phase algorithm** to restructure loop iterations efficiently.

- **Initiation Interval (II):**
  - Minimum cycles between initiation of loop iterations.
  - Determined by dependency analysis and resource constraints.
- **Scheduling**:
  - Instructions from different iterations are interleaved based on the II.
- **Kernel Formation:**
  - The core repeating body of the loop where overlapping is maximized.
- **Example**:
  - LLVM computes II and interleaves loop bodies to fill execution slots with minimal idle time.

SOFTWARE PIPELINING EXAMPLE

Illustrating overlapping execution using a simple loop.

Original loop:

```
for (int i = 0; i < n; i++)
{
    tmp = A[i] + B[i];
    C[i] = tmp * D[i];
}
```

Pipelined execution:

- **Prologue**: `Load A[0], B[0], D[0]`
- **Kernel**: `Load A[1], B[1], D[1]; compute/store C[0]`
  `Load A[2], B[2], D[2]; compute/store C[1]`
- **Epilogue**: compute/store final elements

Benefit:

- Maximizes pipeline utilization and hides latency.
- Overlapped loads, compute, and stores across iterations.
- Better functional unit utilization.

INSTRUCTION SCHEDULING COMPLEXITY

Instruction scheduling is an **NP-complete problem** due to the need to explore all valid orderings under constraints.

Challenges:

- Data dependencies.
- Functional unit availability.
- Instruction latencies and resource conflicts.

Implication:

- Optimal solutions are computationally expensive.
- Heuristics are used in practice.

OPTIMAL vs. HEURISTIC APPROACHES

Comparison of ideal versus practical methods for scheduling.

Optimal scheduling:

- Uses exhaustive search.
- Guarantees best possible performance.
- Exponential complexity.

Heuristic scheduling:

- Uses rules (e.g., list scheduling: ranks by readiness and critical path length, priority-based chooses earliest executable instructions).
- Much faster, near-optimal.

Method	Complexity	Quality
Exhaustive Search	Exponential	Optimal
List Scheduling	Polynomial	~90-95% optimal
Priority-Based	Polynomial	~85-90% optimal

IMPACT OF REGISTER PRESSURE ON SCHEDULING

High register pressure complicates scheduling and often requires spilling, which adds overhead and delays.

Problem:

- When too many variables are live, registers run out and values must be saved to memory.

Solution:

- The scheduler tracks live ranges and considers spill costs to avoid unnecessary spills.
- May delay or reorder instructions to keep register usage low.

Example:

- LLVM integrates register pressure metrics into its scheduling decisions to improve balance between performance and resource limits.

INFLUENCE OF TARGET INSTRUCTION SET ARCHITECTURE (ISA)

The target ISA defines how instructions are structured and impacts which scheduling strategies are effective.

Effects:

- Instruction formats, latency, parallelism, and supported addressing modes all influence scheduling.

RISC-V Example:

- Simpler instruction set reduces scheduling complexity.
- Lack of complex addressing modes requires explicit load/store sequencing.

Implication:

- LLVM uses target-specific models to fine-tune instruction placement according to ISA features.

SCHEDULING AND ENERGY EFFICIENCY

Instruction scheduling isn’t only about speed—it also affects energy efficiency.

Energy-Aware Scheduling:

- Prefer low-power units when possible.
- Avoid toggling between unit types (e.g., ALU to FPU back to ALU).

Use Cases:

- Embedded systems, mobile processors, IoT devices.

LLVM Support:

- Schedulers for embedded targets use energy profiles to guide decisions.

SCHEDULING FOR VECTOR AND SIMD

SIMD/vector hardware imposes alignment, width, and data layout constraints.

Challenge:

- Align operations to vector lanes, manage packed data.

Scheduling Goal:

- Group scalar operations into vectorizable patterns.
- Align memory accesses to vector boundaries.

Example:

- LLVM's vectorization passes (like **SLP** and **Loop Vectorizer**) collaborate with the scheduler to enable efficient SIMD execution.

COMPILER BACKEND STRUCTURE AND SCHEDULING

Instruction scheduling is one stage in the compiler backend, coordinated with selection and register allocation.

Pipeline Overview:

- IR optimization → instruction selection → scheduling → register allocation → machine code emission.

Role:

- Scheduler bridges abstract IR and hardware-specific register usage.

LLVM View:

- Schedulers in LLVM are placed before register allocation and adjust for latency, pressure, and target constraints.

REOPTIMIZATION AFTER INSTRUCTION SELECTION

Post-selection transformations can alter instruction sequences, requiring re-scheduling.

Causes:

- Peephole optimizations and rewrites change instruction dependencies.
- Register allocation may reintroduce stalls.

Strategy:

- Compilers reinvoke the scheduler to clean up or re-optimize instruction order.

LLVM Example:

- The MachinePeephole pass triggers local re-scheduling to maintain optimality.

SCHEDULING OPTIMIZATIONS FOR RISC-V

RISC-V's simplicity and modularity guide specific scheduling tactics.

Characteristics:

- Load/store architecture
- Fewer instruction formats, consistent latency

LLVM Adaptations:

- Aggressive use of register renaming due to abundant registers
- Precise latency modeling for loads and branches

Example:

- Schedule multiply-add pairs using fused multiply-add (FMA) if hardware supports it.

ADVANCED SCHEDULING TECHNIQUES

Beyond classical methods, modern compilers explore hybrid and speculative scheduling.

Hybrid methods:

- Combine global + local + AI-based strategies

Speculative scheduling:

- Moves instructions ahead of branches assuming predicted outcomes

Runtime feedback:

- Profile-guided scheduling using actual execution data

Example:

- LLVM's MachineScheduler dynamically selects strategies per target

AI AND INSTRUCTION SCHEDULING

AI is increasingly used to enhance instruction scheduling beyond traditional heuristics.

Techniques:

- Reinforcement learning (e.g., MLGO by Google).
- Graph neural networks (e.g., Facebook's ProGraML).

Benefits:

- Adaptive to architecture and workload.
- Learns from experience and profiling data.

Example:

- LLVM enhanced with MLGO learns optimal schedules for specific performance goals.

CURRENT AI TOOLS IN COMPILATION

Several AI-based tools are already used in compilers to enhance instruction scheduling and other optimization tasks.

MLGO (Google) (Machine Learning Guided Compiler**)**

- Uses reinforcement learning to guide LLVM scheduling.
- Learns from real-world compilation performance.

ProGraML (Facebook) (A Graph-based Program Representation Which uses Machine Learning**):**

- Graph neural network model trained on code graphs.
- Enables structural understanding of instruction flow.

CompilerGym (Facebook) (environments for optimization tasks**)**:

- Environment for benchmarking and training ML models for compiler tasks. Built upon OpenAI Gym, for python, etc.
- Supports training AI agents for instruction ordering, loop unrolling, etc.

CASE STUDY – AI SCHEDULING INTEGRATION

Scenario: ML-guided selection of instruction order for matrix multiplication loop.

Baseline:

- Static scheduling introduces redundant stalls due to poor latency hiding

ML-optimized:

- Model suggests reordering loads and multiplies to overlap better

Comparison:

Metric	Static	ML-Optimized
Cycles/Loop	120	87
Cache Misses	12	6

- 27.5% improvement in loop execution time.
- Better data locality and resource balance.

AI FUTURE IN SCHEDULING

AI's role in instruction scheduling is expected to grow with more advanced models and tighter integration.

Future capabilities:

- Adaptive schedulers that tailor strategies to architecture + workload.
- Online learning during compilation or execution.

Integration:

- Closer collaboration between compiler passes and AI agents.

Potential:

- Near-optimal schedules with lower complexity.
- Better generalization across codebases.

FUTURE CHALLENGES IN SCHEDULING

Several challenges remain in making instruction scheduling more adaptive, scalable, and portable.

Scalability: Instruction-level optimization grows exponentially with program size

Portability: Hardware-specific heuristics often don't generalize well

Heterogeneous systems: CPUs, GPUs, and accelerators require unified strategies

Data-driven compilation: Need for large, diverse datasets to train ML models effectively

Goal: Develop robust scheduling that adapts across hardware and use cases

REAL-WORLD APPLICATION EXAMPLES

Instruction scheduling impacts many high-performance domains.

- **Embedded systems**: Schedulers ensure tight loops meet timing constraints
- **Scientific computing**: Loop fusion + pipelining optimize large matrix operations
- **Mobile apps**: Trade-offs between energy and latency via smart scheduling
- **Game engines**: Physics simulations scheduled to avoid frame drops
- **Takeaway**: Instruction scheduling is not academic—it affects real outcomes

SUMMARY AND FINAL THOUGHTS

Instruction scheduling is a cornerstone of compiler backends with wide-ranging performance impact.

Recap:

- Covered architectural context, scheduling strategies, software pipelining, and AI integration

Lessons:

- Hardware-aware scheduling yields significant performance gains
- Heuristics and AI complement each other

Outlook:

- More adaptive, explainable, and architecture-specific scheduling will define next-generation compilers

See you at the next Lecture!

Don’t forget to consult the Q&A blog on Brightspace!