PDF3 - Paralleism

Understanding the differences between coarse-grained, fine-grained, simultaneous multithreading (SMT), and superscalar architectures involves exploring their distinct approaches to processing instructions and managing computational resources. Here are the key points for each:

Coarse-Grained Multithreading

Advantages: 1. **Simplicity:** Easier to implement compared to fine-grained and SMT. 2. **Reduced Context Switching Overhead:** Since the switching happens at a higher granularity, there's less frequent context switching which can reduce overhead. 3. **Effective Use of Pipeline:** When one thread stalls, another can be executed, thus keeping the pipeline busy.

Disadvantages: 1. **Lower Resource Utilization:** During thread switching, resources might be underutilized as the switching happens at the end of a process or block of instructions. 2. **Latency:** Higher latency in switching between threads since it happens less frequently compared to fine-grained multithreading.

Fine-Grained Multithreading

Advantages: 1. **Higher Resource Utilization:** By switching between threads more frequently (often every clock cycle), resources are used more efficiently. 2. **Reduced Latency in Pipeline Stalls:** Frequent switching helps in hiding latencies due to pipeline stalls, thus keeping the execution units busy.

Disadvantages: 1. **Complex Implementation:** Requires more complex hardware to manage the frequent switching. 2. **Increased Overhead:** More frequent context switching can increase the overhead.

Simultaneous Multithreading (SMT)

Advantages: 1. **Maximizes Resource Usage:** Allows multiple threads to be executed simultaneously within a single CPU core, leading to higher utilization of CPU resources. 2. **Improved Throughput:** Can significantly increase the throughput of the system as multiple instructions from different threads are executed in parallel. 3. **Latency Hiding:** Similar to fine-grained multithreading, SMT can hide latencies more effectively.

Disadvantages: 1. **Complexity:** Very complex to implement due to the need to manage multiple threads and ensure they don't interfere with each other. 2. **Resource Contention:** Threads may compete for the same resources, leading to potential contention and reduced performance. 3. **Security Concerns:** Increased complexity can lead to security vulnerabilities, such as side-channel attacks (e.g., Spectre and Meltdown).

Superscalar

Advantages: 1. **Instruction-Level Parallelism:** Can execute multiple instructions simultaneously, increasing the instruction throughput. 2. **Higher Performance:** By dispatching multiple instructions to different execution units in a single cycle, performance is significantly enhanced. 3. **Dynamic Scheduling:** Can dynamically schedule instructions out of order to maximize resource usage and minimize stalls.

Disadvantages: 1. **Complexity and Cost:** The hardware required to manage multiple pipelines and handle out-of-order execution is complex and expensive. 2. **Power Consumption:** Higher power consumption due to increased hardware complexity. 3. **Diminishing Returns:** Beyond a certain point, adding more execution units yields diminishing returns due to instruction dependencies and the difficulty of finding sufficient parallelism.

Summary

- **Coarse-Grained Multithreading**: Simpler, lower resource utilization, less frequent switching.
- **Fine-Grained Multithreading**: Higher resource utilization, frequent context switching, more complex.

- **Simultaneous Multithreading (SMT)**: Maximizes resource usage, complex, potential for resource contention and security issues.
- **Superscalar**: High instruction throughput, complex and costly, diminishing returns beyond a certain level of parallelism.

Each approach has its own set of trade-offs and is suitable for different types of applications and workloads.

Thread-level parallelism (TLP) and instruction-level parallelism (ILP) are two strategies for improving the performance of processors by parallelizing workloads, but they operate at different levels and have distinct characteristics. Here's a detailed comparison of TLP and ILP, including their advantages and disadvantages:

Thread-Level Parallelism (TLP)

Definition: TLP refers to the ability of a processor to execute multiple threads or processes concurrently. This can be achieved through techniques such as multithreading (e.g., SMT, coarse-grained, fine-grained) and multi-core processors.

Advantages: 1. **Higher Throughput:** Multiple threads can be executed simultaneously, increasing overall system throughput. 2. **Better Resource Utilization:** By running multiple threads, processors can keep more of their execution units busy, especially when some threads are stalled due to waiting for data or other resources. 3. **Latency Tolerance:** TLP can help hide latencies by switching to other threads when one is waiting for I/O operations or memory access. 4. **Scalability:** Multi-core architectures can be scaled by adding more cores to handle additional threads.

Disadvantages: 1. **Complexity:** Managing multiple threads requires complex hardware and software support, including scheduling, synchronization, and context switching. 2. **Resource Contention:** Multiple threads might compete for shared resources such as caches, memory, and I/O bandwidth, leading to contention and potential performance degradation. 3. **Overhead:** Thread management introduces overhead in terms of context switching and synchronization, which can affect performance if not managed efficiently.

Instruction-Level Parallelism (ILP)

Definition: ILP refers to the ability of a processor to execute multiple instructions from a single thread simultaneously. This is typically achieved through techniques like pipelining, superscalar execution, and out-of-order execution.

Advantages: 1. **Increased Instruction Throughput:** By executing multiple instructions concurrently, ILP can significantly improve the performance of a single thread. 2. **Efficient Use of Execution Units:** Techniques like superscalar execution and out-of-order execution make effective use of the processor's execution units, minimizing idle time. 3. **Reduced Latency:** ILP techniques can help reduce the execution time of individual instructions, leading to lower overall latency for a single thread.

Disadvantages: 1. **Complex Hardware:** Implementing ILP requires sophisticated hardware mechanisms, such as multiple execution units, complex scheduling logic, and dependency checking, which increases design complexity and cost. 2. **Diminishing Returns:** Beyond a certain point, adding more parallel execution units yields diminishing returns due to instruction dependencies and limited parallelism in the instruction stream. 3. **Power Consumption:** The additional hardware required for ILP can lead to higher power consumption, which is a critical consideration for modern processors.

Summary

- **TLP (Thread-Level Parallelism)**
  - **Focus:** Multiple threads/processes.
  - **Techniques:** Multithreading (SMT, coarse-grained, fine-grained), multi-core processors.
  - **Advantages:** Higher throughput, better resource utilization, latency tolerance, scalability.
  - **Disadvantages:** Complexity, resource contention, overhead.
- **ILP (Instruction-Level Parallelism)**
  - **Focus:** Multiple instructions from a single thread.

  - **Techniques:** Pipelining, superscalar execution, out-of-order execution.
  - **Advantages:** Increased instruction throughput, efficient use of execution units, reduced latency.
  - **Disadvantages:** Complex hardware, diminishing returns, power consumption.

Practical Use Cases

- **TLP** is highly effective in environments where workloads can be divided into independent threads, such as server applications, parallel computing tasks, and multitasking operating systems.
- **ILP** is beneficial for improving the performance of individual applications and programs, particularly those that cannot easily be parallelized at the thread level, such as many legacy applications and single-threaded workloads.

Both TLP and ILP are essential for modern processors, and they are often used in combination to achieve optimal performance across a wide range of applications and workloads.

Flynn Taxonomy (1966)

SISD: Single Instruction Single Data Uniprocessor systems

MISD: Multiple Instruction Single Data. No practical configuration and no commercial systems

SIMD: Single Instruction Multiple Data

Simple programming model, low overhead, flexibility, custom integrated circuits

MIMD: Multiple Instruction Multiple Data Scalable, fau## PDF3 - Paralleism

PDF 4, 5 Branch Hazards and Static Branch Prediction Technique

Conditional Branch Instruction: the branch is taken only if the condition is satisfied. The branch target address (BTA) is stored in the Program Counter (PC) instead of the address of the next instruction in the sequential instruction stream

Execution of conditional branches

stage MIPS pipeline

- Branch Outcome and Branch Target Address are ready at the end of the EX stage (3 rd stage)
- Conditional branches are solved when PC is updated at the end of the ME stage (4 th stage)

Control hazards reduce the performance from the ideal speedup gained by the pipelining since they can make it necessary to stall the pipeline

Branch Hazards

To feed the pipeline we need to fetch a new instruction at each clock cycle, but the branch decision (to change or not change the PC) is taken during the MEM stage. This delay to determine the correct instruction to fetch is called Control Hazard or Conditional Branch Hazard

Forwarding

Data forwarding uses temporary results stored in the pipeline registers instead of waiting for the write back of results in the RF. We need to add multiplexers at the inputs of ALU to fetch inputs from pipeline registers to avoid the insertion of stalls in the pipeline.

Early Evaluation of the PC

MIPS processor compares registers, computes branch target address and

With the branch decision made during ID stage, there is a reduction of the cost associated with each branch (branch penalty) : We need only one clock cycle stall after each branch Or a flush of only one instruction following the branch

Delayed Branch Technique

The MIPS compiler always schedules a branch independent instruction after the branch. The behavior of the delayed branch is the same whether or not the branch is taken.

I understand. Here's the table summarizing the three methods used to fill the branch delay slot: from the target of the branch, from the fall-through path, and from before the branch:

Method	Description	Advantages	Disadvantages
From Target of the Branch	Move an instruction from the branch target into the delay slot.	Efficient if the branch is taken; reduces wasted cycles.	Ineffective if the branch is not taken; can complicate pipeline management.
From Fall-Through Path	Move an instruction from the fall-through path (next sequential instruction) into the delay slot.	Utilizes the slot when the branch is not taken.	Ineffective if the branch is taken; needs careful selection to avoid hazards.
From Before the Branch	Move an instruction that appears before the branch into the delay slot.	Simple and straightforward, no additional instructions needed.	Limited by dependencies and ordering constraints.

Summary:

- **From Target of the Branch:** Best if the branch is frequently taken; may cause issues if not taken.
- **From Fall-Through Path:** Useful when the branch is rarely taken; limited use if the branch is usually taken.
- **From Before the Branch:** Simple to implement; constrained by instruction dependencies and ordering.

PDF 6 Dynamic Branch Prediction

Dynamic Branch Prediction

Schemes: Dynamic branch prediction is based on two interacting mechanisms:

Branch Outcome Predictor:

To predict the direction of a branch (i.e. taken or not taken). * Branch Target Predictor: To predict the branch target address in case of taken branch.

These modules are used by the Instruction Fetch Unit to predict the next instruction to read in the I cache.

If branch is not taken PC is incremented.

If branch is taken BTP gives the target address

Branch Target Buffer

Branch Target Buffer (Branch Target Predictor) is a cache storing the predicted branch target address for the next instruction after a branch • We access the BTB in the IF stage using the instruction address of the fetched instruction (a possible branch) to index the cache.

The predicted target address is expressed as PC relative

Branch History Table

Branch History Table (or Branch Prediction Buffer):

Table containing 1 bit for each entry that says whether the branch was recently taken or not .

Table indexed by the lower portion of the address of the branch instruction .

Prediction : hint that it is assumed to be correct , and fetching begins in the predicted direction .

If the hint turns out to be wrong , the prediction bit is inverted and stored back. The pipeline is flushed and the correct sequence is executed .

The table has no tags (every access is a hit) and the prediction bit could has been put there by another branch with the same low

order address bits: but it does not matter . The prediction is just a hint

A misprediction occurs when The prediction is incorrect for that branch , or

The same index has been referenced by two different branches , and the previous history refers to the other branch

To solve this problem it is enough to increase the number of rows in the BHT or to use a hashing function (such as in GShare)

2 bit Branch History Table

The prediction must miss twice before it is changed.

In a loop branch, at the last loop iteration, we do not need to change the prediction.

For each index in the table, the 2 bits are used to encode the four states of a finite state machine

n-bit Branch History Table

• Generalization: n-bit saturating counter for each entry in the prediction buffer.

The counter can take on values between 0 and 2^n-1

When the counter is greater than or equal to one-half of its maximum value (2^n-1), the branch is predicted as taken.

Otherwise, it is predicted as untaken. • As in the 2-bit scheme, the counter is incremented on a taken branch and decremented on an untaken branch. • Studies on n-bit predictors have shown that 2-bit predictors behave almost as well.

Correlating Branch Predictors

BHT predictors use only the recent behavior of a single branch to predict the future behavior of that branch.

Basic Idea: the behavior of recent branches are correlated, that is the recent behavior of other branches rather than just the current branch (we are trying to predict) can influence the prediction of the current branch.

Branch predictors that use the behavior of other branches to make a prediction are called Correlating Predictors or 2-level Predictors.

Example a (1,1) Correlating Predictors means a 1-bit predictor with 1-bit of correlation: the behavior of last branch is used to choose among a pair of 1-bit branch predictors.

I skipped a lot of parts that are maybe important

GA Predictor

The Branch History Table (BHT) and Global History Register with Global Prediction (GAg) are both techniques used for dynamic branch prediction in computer architecture, but they operate differently and have distinct characteristics. Here's a comparison between the two:

Branch History Table (BHT)

Description: - A BHT is a simple table used for branch prediction where each entry in the table corresponds to a branch instruction. It records the outcomes (taken or not taken) of recent executions of that branch.

Operation: - Each entry typically consists of a counter or a set of bits that are updated based on the actual outcome of the branch instruction. For example, a 2-bit counter can represent four states to indicate how likely a branch is to be taken.

Advantages: - **Simplicity:** Easy to implement and understand. - **Local Prediction:** Works well for branches that have a consistent behavior pattern.

Disadvantages: - **Limited Context:** Only considers the history of individual branches, which may not capture complex patterns or correlations with other branches. - **Storage Requirements:** Requires a table entry for each branch, which can be large for programs with many branches.

Global History Register with Global Prediction (GAg)

Description: - GAg uses a global history register (GHR) that records the outcomes of the most recent branches (usually as a shift register) and a global pattern history table (PHT) indexed by this global history.

Operation: - The GHR is a single register that captures the history of the last N branches (regardless of which branch instruction they were). - The PHT is indexed using the GHR, and it stores the prediction for the combination of recent branch outcomes. - This method leverages the global branch history to make predictions, capturing more complex patterns across different branches.

Advantages: - **Global Context:** Takes into account the outcomes of multiple branches, potentially improving prediction accuracy for branches that are correlated. - **Captures Complex Patterns:** More effective for programs where branch outcomes are interdependent.

Disadvantages: - **Complexity:** More complex to implement than a simple BHT. - **Longer History Management:** Managing and updating the global history register can be more challenging. - **Potential for Conflicts:** Multiple branches may map to the same entry in the PHT, causing aliasing and reducing prediction accuracy.

Summary Table

Aspect	Branch History Table (BHT)	Global History Register with Global Prediction (GAg)
History Type	Local to each branch	Global across all branches
Storage	Table indexed by branch address	Global History Register (GHR) and Pattern History Table (PHT)
Prediction Basis	Recent outcomes of individual branches	Combined recent outcomes of multiple branches

Aspect	Branch History Table (BHT)	Global History Register with Global Prediction (GAg)
Complexity	Simple	More complex
Accuracy	Good for independent branches	Higher for interdependent branches
Implementation	Straightforward	Requires management of global history and handling potential aliasing
Advantages	Simple, easy to implement, effective for consistent branch patterns	Captures global patterns, potentially more accurate for correlated branches
Disadvantages	Limited context, large storage for many branches	Complex, potential aliasing issues, more challenging to implement

In essence, while BHT focuses on individual branch histories, GAg leverages a global approach to capture broader execution patterns, which can improve prediction accuracy in scenarios with interdependent branches.

PDF 7 - Instruction Level Parallelism

Pipeline performance • Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls

Ideal pipeline CPI measure of the maximum performance attainable by the implementation
*** Structural hazards** HW cannot support this combination of instructions
*** Data hazards** Instruction depends on result of prior instruction still in the pipeline
*** Control hazards** Caused by delay between the fetching of instructions and decisions about changes in control flow (branches, jumps, exceptions)

Hazards limit performance * Structural : need more HW resources * Data : need forwarding, compiler scheduling * Control : early evaluation & PC, delayed branch, predictors

Complex Pipelining

Pipelining becomes complex when we want high performance in the presence of: Long latency or partially pipelined floating

point units Multiple function and memory units Memory systems with variable access time Precise exception

Issues in Complex Pipeline Control

- Structural conflicts at the execution stage if some FPU or memory unit is not pipelined and takes more than one cycle Structural conflicts at the write
- back stage due to variable latencies of different functional units
- Out of order write hazards due to variable latencies of different FUs

Complex In Order Pipeline

Delay writeback so all operations have same latency to WB stage Instructions commit in order, simplifies precise exception implementation

The following checks need to be made before the Issue

stage can dispatch an instruction > Is the required function unit available? > Is the input data available? RAW > Is it safe to write the destination? WAR/WAW > Is there a structural conflict at the WB stage?

Getting higher performance

In a pipelined machine, actual CPI is derived as:

`CPIpipe=CPIideal+Structural_stalls+Data_hazard_stalls+Control_stalls`

Reduction of any right-hand term reduces CPI pipe (increase Instructions Per Clock–IPC)

Technique to increase CPI pipe could create further problems with hazards

To reach higher performance (for a given technology) – more parallelism must be extracted from the program

Dependences must be detected and solved, and instructions must be ordered (scheduled) so as to achieve highest parallelism of execution compatible with available resources.

If two instructions are dependent, they cannot execute in parallel: they must be executed in order or only partially overlapped

Name Dependences

when 2 instructions use the same register or memory location (called name), but there is no flow of data between the instructions associated with that name

> **Antidependence** when j writes a register or memory location that instruction i reads. The original instructions ordering must be preserved to ensure that i reads the correct value.
>
> Output Dependence : when i and j write the same register or memory location. The original instructions ordering must be preserved to ensure that the value finally written corresponds to j.

- Name dependences are not true data dependences, since there is no value (no data flow) being transmitted between instructions.
- If the name (register number or memory location) used in the instructions could be changed, the instructions do not conflict.
- Dependences through memory locations are more difficult to detect (“ memory disambiguation ” problem), since two addresses may refer to the same location but can look different.
- Register renaming can be more easily done.
- Renaming can be done either statically by the compiler or dynamically by the hardware.

Data Dependences and Hazards

A data/name dependence can potentially generate a data hazard (RAW, WAW, or WAR), but the actual hazard and the number of stalls to eliminate the hazards are a property of the pipeline. * RAW hazards correspond to true data dependences.

- WAW hazards correspond to output dependences
- WAR hazards correspond to antidependences

Dependences are a property of the program, while hazards are a property of the pipeline

Control Dependences

determines the ordering of instructions

- Instructions execution in program order to ensure that an instruction that occurs before a branch is executed before the branch.
- Detection of control hazards to ensure that an instruction (that is control dependent on a branch) is not executed until the branch direction is known.

control dependence is not the critical property that must be preserved.

Program Properties

Program Properties

Exception behavior : Preserving exception behavior means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program

Data flow: Actual flow of data values among instructions that produces the correct results and consumes them

PDF 8 - Static Scheduling and VLIW Architectures

Dynamic Scheduling: Depend on the hardware to locate parallelism

Hardware intensive approaches dominate desktop and server markets

Static Scheduling: Rely on software for identifying potential parallelism

Dynamic Scheduling

Basically: Instructions are fetched and issued in program order (in-order-issue) * Execution begins as soon as operands are available

possibly, out of order execution

note: possible even with pipelined scalar architectures . * Out-of order execution introduces possibility of **WAR, WAW** data hazards. * Out-of order execution implies out of order completion unless there is a re-order buffer to get in-order completion

Superscalar and *VLIW*

`IdealCPI < 1`

Dynamic scheduling

Pipelining Initial goal

`IdealCPI = 1`

Sequential

`IdealCPI > 1`

VLIW

- fixed number of instructions (4-16)
- **scheduled by the compiler;** put ops into wide templates
- Style: " Explicitly Parallel Instruction Computer (EPIC)

  Processor can initiate multiple operations per cycle

  Low hardware complexity

  Explicit parallelism

  Single control flow

VLIW processors

- **Operation:** is a unit of computation (add, load, branch = instruction in sequential ar.)
- **Instruction:** set of operations that are intended to be issued simultaneously

All operations that are supposed to begin at the same time are packaged into a single VLIW instruction

Each operation slot is for a fixed function

Constant operation latencies are specified

Architecture requires guarantee of:

- Parallelism within an instruction => no x-operation RAW check
- No data use before data ready => no data interlocks

VLIW Compiler Responsibilities

The compiler:

Schedules to maximize parallel execution

Exploit ILP and LLP (Loop Level Parallelism)

It is necessary to map the instructions over the machine functional units

- This mapping must account for time constraints and dependencies among the tasks

Guarantees intra * instruction parallelism

Schedules to avoid data hazards (no interlocks)

- Typically separates operations with explicit NOPs

The goal is to minimize the total execution time for the program

The compiler: * Schedules to maximize parallel execution > Exploit ILP and LLP (Loop Level Parallelism) > It is necessary to map the instructions over the machine functional units > This mapping must account for time constraints and dependencies among the tasks * Guarantees intra-instruction parallelism * Schedules to avoid data hazards (no interlocks) > Typically separates operations with explicit NOPs * The goal is to minimize the total execution time for the program

Static Scheduling

Try to keep pipeline full (in single issue pipelines) or utilize all FUs in each cycle (in VLIW) as much as possible to reach better ILP and therefore higher parallel speedups.

- Compilers can use sophisticated algorithms for code scheduling to exploit ILP (Instruction Level Parallelism).
- The amount of parallelism available within a basic block is quite small.
- Data dependence can further limit the amount of ILP we can exploit within a basic block to much less than the average basic block size.
- To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks (i.e. across branches).

dependences are avoided by code reordering

Basic Block Definition

a sequence of straight non-branch instructions

Here's the information organized into a table:

VLIW: Pros and Cons	
Pros	**Cons**
- Simple HW	- Huge number of registers to keep active the FUs
- Easy to extend the #FUs	- Needed to store operands and results
- Good compilers can effectively detect parallelism	- Large data transport capacity between: * FUs and register files * Register files and Memory
	- High bandwidth between i-cache and fetch unit
	- Large code size
	- Knowing branch probabilities
	- Profiling requires a significant extra step in build process
	- Scheduling for statically unpredictable branches

Static Scheduling: methods

Simple code motion

Loop unrolling & loop peeling

Software pipeline

Global code scheduling (across basic block) * Trace scheduling * Superblock scheduling * Hyper-block scheduling * Speculative Trace scheduling

Trace scheduling

focuses on traces

A trace is a loop-free sequence of basic blocks embedded in the control flow graph (Fisher)

It is an execution path which can be taken for some set of inputs

The chances that a trace is actually executed depends on the input set that allows its execution

Some traces are executed much more frequently than others

- Pick string of basic blocks, a trace, that represents most frequent branch path
- Use profiling feedback or compiler heuristics to find common branch paths
- Schedule whole "trace" at once
- Add fixup code to cope with branches jumping out of trace

Trace scheduling and loops

- Trace scheduling cannot proceed beyond a loop barrier
- Techniques used to overcome this limitation are based on loop unrolling #### Negative effects on unrolling
- Unrolling produces much extra code

- It also looses performance, because of the costs of starting and closing the iterations #### Traces scheduling schedules traces in order of decreasing probability of being executed
- So, most frequently executed traces get better schedules
- Traces are scheduled as if they were basic blocks (no special considerations for branches)

  Trace: a sequence of instructions which may Include branches but not including loops

PDF 9

Code Motion in Trace Scheduling

In addition to need of compensation codes there are restrictions on movement of a code in a trace:

- The dataflow of the program must not change
- The exception behavior must be preserved #### Dataflow can be guaranteed to be correct by maintaining two dependencies:
- Data dependency
- Control dependency

There are two solutions to eliminate control dependency:

- By use of predicate instructions (Hyperblock scheduling) and removing the branch.
- By use of speculative instructions (Speculative Scheduling) and speculatively move an instruction before the branch.

Technique	Predicate Instructions (Hyperblock Scheduling)	Speculative Instructions (Speculative Scheduling)
Description	Uses predicate registers to conditionally execute instructions, reducing branches	Executes instructions before branch outcome is known based on predictions
Mechanism	Combines multiple basic blocks into a hyperblock, guarded by predicates	Moves instructions before branches, speculating they will be needed
Branch Handling	Minimizes branches by conditional execution within hyperblocks	Reduces idle cycles by pre-executing instructions based on branch prediction
Execution Control	Instructions are executed based on the evaluation of predicate conditions	Instructions are executed speculatively, assuming the prediction is correct
Impact on Performance	Enhances parallelism by reducing branch penalties	Improves utilization of execution units by overlapping instruction execution with branch decision
Compiler/Processor Role	Compiler organizes instructions into hyperblocks with predicates	Compiler/processor schedules instructions speculatively
Examples of Use	If-else statements are converted into predicated instructions within a hyperblock	Instructions after a branch are moved before the branch for speculative execution
Benefits	Efficient use of functional units, less branch disruption	Better performance by keeping execution units busy, minimizing idle time
Drawbacks	Increased complexity in scheduling and handling predicates	Risk of executing unnecessary instructions if the speculation is incorrect

This table highlights the key aspects and benefits of using predicate instructions with hyperblock scheduling and speculative instructions with speculative scheduling in VLIW architectures.

Speculation and prediction are closely related concepts, but they refer to different aspects of handling uncertain outcomes in computing, particularly in processor architecture.

Prediction

- **Definition:** Prediction is the process of making an educated guess about the outcome of a future event, such as the direction of a branch in a program (e.g., whether an `if` condition will be true or false).
- **Purpose:** The main goal of prediction is to anticipate the direction a branch will take so that the processor can prepare to execute the appropriate instructions.
- **Example:** Branch prediction algorithms in CPUs predict whether a branch will be taken or not taken based on historical data and patterns observed in previous executions.

Speculation

- **Definition:** Speculation is the process of executing instructions based on the prediction made. It involves carrying out the predicted path before the actual outcome is known.
- **Purpose:** The main goal of speculation is to keep the processor's execution units busy and improve performance by executing instructions in advance of the actual decision point.
- **Example:** Once a branch predictor guesses that a branch will be taken, speculative execution will proceed to execute the instructions on that predicted path.

Relationship and Differences

- **Prediction vs. Speculation:** Prediction is about making a guess, while speculation is about acting on that guess. Prediction feeds into speculation; without prediction, there is no basis for speculation.
- **Outcome Handling:**
  - **Prediction:** The outcome of the prediction is a guess (e.g., "Branch will be taken").
  - **Speculation:** The result of speculation is the execution of instructions based on that guess. If the prediction is correct, the speculative execution is beneficial. If incorrect, the speculatively executed instructions are discarded, and the correct path is executed.
- **Risk and Reward:**
  - **Prediction:** Has no inherent risk or direct reward by itself. It simply provides a forecast.
  - **Speculation:** Involves risk because it acts on the prediction. The reward is improved performance if the prediction is correct, and the risk is wasted cycles and potential rollback if the prediction is wrong.

Summary with Example

Consider a branch in a program:

```
if (condition) {
    // block A
} else {
    // block B
}
```

1. **Prediction:** The branch predictor guesses whether `condition` will be true or false (e.g., predicts `condition` is true, so block A will execute).
2. **Speculation:** Based on this prediction, speculative execution begins executing the instructions in block A before knowing if `condition` is actually true.

If the prediction is correct: - The speculative execution of block A is validated, leading to a performance gain because the processor did not wait to determine the condition.

If the prediction is incorrect: - The speculative execution of block A is discarded, and the processor must then execute block B, which incurs a performance penalty due to the misprediction and wasted cycles.

In essence, prediction provides the guess, and speculation takes action based on that guess.

Predicated Execution

Problem: Mispredicted branches limit ILP Solution: Eliminate hard to predict branches with predicated execution

- Almost all IA-64 instructions can be executed conditionally under predicate
- Instruction becomes NOP if predicate register false

Rotating Register Files

Rotating Register Files (RRFs)

Rotating Register Files (RRFs) are a specialized type of register file architecture used primarily in very long instruction word (VLIW) and explicitly parallel instruction computing (EPIC) processors. They are designed to support software-pipelined loops efficiently by providing a mechanism for registers to be reused across iterations without explicit renaming or handling by the compiler or hardware.

Key Concepts

1. **Registers and Loop Iterations:**
   - In software-pipelined loops, different iterations of a loop can be executing simultaneously at different stages. Each iteration requires its own set of registers to hold temporary variables.
   - RRFs simplify this by allowing a fixed set of physical registers to be "rotated" or reused across iterations.
2. **Rotation Mechanism:**
   - The register file is logically divided into several "windows" or "frames." Each window corresponds to the register set used by a particular loop iteration.
   - As a new iteration begins, the register file rotates so that the same physical registers are mapped to the new iteration's logical registers.
3. **Benefits of RRFs:**
   - **Efficient Register Usage:** By rotating the register set, RRFs avoid the need for an excessive number of physical registers, which would be required if each iteration had its own distinct set of registers.
   - **Simplified Register Management:** Compilers and hardware can handle software-pipelined loops more efficiently without complex register renaming mechanisms.
   - **Reduced Overhead:** The rotation mechanism reduces the overhead of saving and restoring registers across iterations, leading to improved performance.

Example Scenario Consider a loop that is being software-pipelined, with three stages of execution (e.g., fetch, execute, write-back):

```
for (int i = 0; i < N; i++) {
    // Stage 1: Load data
    // Stage 2: Perform computation
    // Stage 3: Store results
}
```

Using RRFs, the register file might be divided into three windows, one for each stage of the loop:

1. **Iteration 1:**
   - Stage 1 uses registers R0-R3.
   - Stage 2 uses registers R4-R7.
   - Stage 3 uses registers R8-R11.
2. **Iteration 2 (after rotation):**
   - Stage 1 now uses registers R4-R7.

- Stage 2 uses registers R8-R11.
- Stage 3 uses registers R0-R3.

3. **Iteration 3 (after another rotation):**
   - Stage 1 now uses registers R8-R11.
   - Stage 2 uses registers R0-R3.
   - Stage 3 uses registers R4-R7.

Implementation in VLIW/EPIC Processors VLIW and EPIC architectures rely on compilers to detect and exploit instruction-level parallelism. RRFs are particularly useful in these architectures because they help manage the complexity of parallel execution across multiple loop iterations. By automating the rotation of register sets, RRFs allow the compiler to focus on optimizing instruction scheduling and parallelism without worrying about register allocation for each iteration.

Summary

Rotating Register Files (RRFs): - Provide an efficient way to reuse registers across loop iterations in software-pipelined loops. - Simplify register management by using a rotation mechanism. - Enhance performance in VLIW and EPIC architectures by reducing the overhead of register saving and restoring.

This architectural feature is crucial for achieving high performance in processors that rely heavily on parallel execution and efficient loop handling.

Dynamic scheduling

A Data Structure for Correct Issues Keeps track of the status of Functional Units

CDC6600 Scoreboard

Instructions dispatched in-order to functional units provided no structural hazard or WAW * Stall on structural hazard, no functional units available

- Only one pending write to any register

Instructions wait for input operands (RAW hazards) before execution

Can execute out-of-order

Instructions wait for output register to be read by preceding instructions (WAR)

Result held in functional unit until register free

Enables out-of-order execution and completion (commit)

Out-of order execution introduces possibility of WAR, WAW data hazards.

Scoreboard centralizes hazard management

- Every instruction goes through the scoreboard
- Scoreboard determines when the instruction can read its operands and begin execution
- Monitors changes in hardware and decides when a stalled instruction can execute
- Controls when instructions can write results

Certainly! Let's compare the new scoreboard-based pipeline to a traditional (or old) pipeline. We'll outline the stages and characteristics of each, and highlight the differences in their approach to handling hazards and scheduling.

Old Pipeline Stages

A traditional pipeline typically follows a simple, linear sequence of stages, such as the classic RISC (Reduced Instruction Set Computing) pipeline.

Stage	Description
Fetch (IF)	The instruction is fetched from memory.
Decode (ID)	The instruction is decoded to determine the operation and the operands required.
Execute (EX)	The instruction is executed by the appropriate functional unit.
Memory (MEM)	If the instruction involves memory access, the address is calculated, and the data is read or written.
Write Back (WB)	The result of the execution or memory operation is written back to the register file.

New Pipeline Stages in Scoreboarding

The scoreboard-based pipeline has stages that specifically address the dynamic scheduling and hazard management:

Stage	Description
Issue	The instruction is issued from the instruction queue if the required functional unit (FU) is available and no WAW (Write After Write) hazards exist. The instruction is then sent to the designated FU.
Read Operands	The instruction waits until its source operands are available (i.e., no RAW (Read After Write) hazards). Once the operands are available, they are read and the instruction is ready to execute.
Execute	The instruction is executed by the functional unit. The duration of this stage depends on the type of operation (e.g., addition, multiplication).
Write Result	The instruction waits until there are no WAR (Write After Read) hazards. Once safe, the result is written back to the register file, and the functional unit is marked as available for new instructions.

more detailed explanation in PDF 10

Comparison Table

Aspect	Old Pipeline	New Scoreboarding Pipeline
Stages	Fetch, Decode, Execute, Memory, Write Back	Issue, Read Operands, Execute, Write Result
Hazard Management	Static scheduling, stalls introduced by control unit	Dynamic scheduling, managed by scoreboard
Structural Hazards	Handled by stalling or pipelining functional units	Handled dynamically by scoreboard
Data Hazards	Handled by stalling, forwarding, and pipeline interlocks	Handled by checking RAW, WAR, and WAW hazards dynamically
Control Hazards	Branch prediction, pipeline flushing	Not specifically addressed by scoreboard (depends on other mechanisms)

Aspect	Old Pipeline	New Scoreboarding Pipeline
Execution Parallelism	Limited by static scheduling and stalls	Enhanced by dynamic scheduling, allows for better FU utilization
Flexibility	Less flexible, depends on fixed pipeline stages	More flexible, instructions can progress independently based on resource availability and hazard resolution

Example Scenario

Consider a sequence of instructions:

1. **Old Pipeline:**
   - Instruction fetch (IF) for each instruction occurs in a fixed sequence.
   - Decode (ID) follows, potentially stalling if dependencies are detected.
   - Execution (EX) occurs when operands are available, possibly stalling for RAW hazards.
   - Memory access (MEM) and Write Back (WB) stages follow in sequence.
   - If hazards occur, pipeline stalls are introduced, leading to idle cycles.
2. **New Scoreboarding Pipeline:**
   - Instructions are issued dynamically if the required FU is available (Issue stage).
   - Instructions wait for operands to be available (Read Operands stage) without stalling other instructions.
   - Execution (Execute stage) occurs when the FU and operands are ready.
   - Results are written back when there are no WAR hazards (Write Result stage).
   - Instructions can progress independently, and FUs are kept busy without unnecessary stalls.

Summary

- **Old Pipeline:** Follows a fixed sequence with potential stalls at each stage to handle hazards, leading to less efficient utilization of resources.
- **New Scoreboarding Pipeline:** Uses dynamic scheduling to handle hazards and dependencies, allowing for better parallelism and efficient utilization of functional units.

The scoreboard approach improves performance by reducing idle cycles and better handling data hazards through dynamic scheduling and resource management.

PDF 10 - Dynamic Scheduling: Scoreboard

data dependences that cannot be hidden with bypassing or forwarding cause hardware stalls of the pipeline so we allow instructions behind a stall to proceed

HW rearranges the instruction execution to reduce stalls

Four Stages of Scoreboard Control

Sure! Here's a short summary of the four stages of scoreboard control:

1. **Issue:**
   - Decode instructions and check for structural hazards.
   - Issue instructions in program order if the functional unit (FU) is free and no Write After Write (WAW) hazards exist.
   - Stall if there are structural or WAW hazards until they are cleared.
2. **Read Operands:**
   - Wait until no data hazards (Read After Write, RAW).
   - Read operands from registers once they are available.
   - Instructions can be executed out of order, but no data forwarding occurs.

3. **Execution:**
   - The functional unit begins execution upon receiving operands.
   - Notify the scoreboard when the result is ready.
   - Functional units have specific latency and initiation intervals.
4. **Write Result:**
   - Check for Write After Read (WAR) hazards after execution.
   - If no WAR hazard, write results to the register.
   - Stall if there is a WAR hazard until it is cleared.

   Assume we can overlap issue and write

Scoreboard Implications

- **Solution for WAW:** Detect hazard and stall issue of new instruction until the other instruction completes
- No register renaming
- Need to have multiple instructions in execution phase → Multiple execution units or pipelined execution units
- Scoreboard keeps track of dependences and state of operations

scoreboard structure

Instruction status

Functional Unit status

- **Busy:** Tracks whether an FU is in use.
- **Op:** Specifies the operation being performed by the FU.
- **Fi:** Destination register for the operation's result.
- **Fj, Fk:** Source registers for the operands of the operation.
- **Qj, Qk:** Indicate which FUs are currently producing the values for Fj and Fk.
- **Rj, Rk:** Flags that show if the values in Fj and Fk are ready for use.

These elements collectively help in managing and scheduling the execution of instructions, ensuring that data hazards are resolved and instructions are executed in the correct order.

Register result status

Indicates which functional unit will write each register . Blank if no pending instructions will write that register.

PDF 11 Tomasulo Algorithm

Tomasulo Algorithm Basics

- The control logic and the buffers are distributed with FUs (vs. centralized in scoreboard)
- Operand buffers are called Reservation Stations
- Each instruction is an entry of a reservation station
- Its operands are replaced by values or pointers (Register Renaming)
- Register Renaming allows to:
  - Avoid WAR and WAW hazards
  - Reservation stations are more than registers (so can do better optimizations than a compiler).

- Results are dispatched to other FUs through a Common Data Bus (CDB)
- Load / Stores treated as FUs

Reservation Station Components

- **Tag**: Identifies the specific RS.
- **OP**: The operation to perform.
- **Vj, Vk**: Values of the source operands.
- **Qj, Qk**: Pointers to the RS that will produce the values of Vj and Vk. If zero, the operand value is already in Vj or Vk.
- **Busy**: Indicates if the RS is currently in use.

Note: For each operand, either the value (V-field) or the pointer (Q-field) is valid, not both.

the stages of the Tomasulo algorithm:

1. Issue

- **Instruction Fetching**: An instruction (I) is fetched from the instruction queue.
- **FP Operation Check**: If (I) is a floating-point (FP) operation, check for available reservation stations (RS) to avoid structural hazards.
- **Register Renaming**: Perform register renaming to resolve Write After Read (WAR) hazards.
  - If (I) writes to register (Rx) and instruction (K), which has already been issued, reads (Rx), (K) either already knows the value of (Rx) or knows which instruction will write to it. Hence, the register file (RF) can be linked to (I).
- **Write After Write (WAW) Resolution**: With in-order issue, the RF is linked to (I), ensuring proper sequencing of writes.

2. Execution

- **Operand Availability**: Execute the instruction when both operands are ready.
- **Operand Waiting**: If operands are not ready, monitor the common data bus (CDB) for the results.
- **RAW Hazard Avoidance**: By delaying execution until operands are available, Read After Write (RAW) hazards are avoided. Multiple instructions may become ready simultaneously for the same functional unit (FU).
- **Load and Store Instructions**: These are handled in a two-step process:
  - **Step 1**: Compute the effective address and place it in the load or store buffer.
  - **Loads**: Execute as soon as the memory unit is available.
  - **Stores**: Wait for the value to be stored before sending it to the memory unit.

3. Write-Back (Not detailed in the provided text but commonly included)

- **Result Broadcasting**: The result of the executed instruction is broadcasted on the CDB to all waiting reservation stations and the RF.
- **Updating Buffers**: Update the load/store buffers and mark the RS as available for new instructions.
- **Instruction Completion**: Mark the instruction as completed and remove it from the instruction queue.

These stages ensure that the Tomasulo algorithm effectively handles out-of-order execution while resolving various data hazards dynamically.