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.

side-by-side comparison of the Scoreboard and Tomasulo Algorithm:

Aspect	Scoreboard	Tomasulo Algorithm
Control Logic	Centralized control logic where a central scoreboard manages the execution of instructions. It tracks the status of functional units and manages instruction issue, execution, and completion.	Distributed control logic with Function Units (FUs) having local control for each FU. Each FU has its reservation station to handle instruction issue, execution, and result broadcasting.
Buffers	Centralized reservation stations and a common data bus for communication between functional units. Instructions wait in reservation stations until operands are available.	FU buffers called "reservation stations" with pending operands. Each reservation station holds the instruction, its operands, and status information. This decentralized approach reduces contention and allows for parallel execution.
Register Handling	Instructions reference registers directly, potentially leading to hazards like Write-After-Read (WAR) and Write-After-Write (WAW).	Registers in instructions are replaced by values or pointers to reservation stations (RS). This register renaming technique eliminates register dependencies and hazards, enabling out-of-order execution.
Register Renaming	Not supported in the traditional sense, which can lead to stalls and dependencies on register availability.	Supports register renaming to avoid WAR and WAW hazards. By assigning physical registers dynamically to instructions, the algorithm prevents data hazards and improves instruction throughput.
Optimizations	Limited optimizations due to centralized control and buffers. Dependencies on the scoreboard can limit parallelism and optimization opportunities.	More reservation stations than physical registers, enabling out-of-order execution and dynamic scheduling. This allows for advanced optimizations that compilers cannot achieve, leading to improved performance.
Data Flow	Results pass through registers, potentially causing bottlenecks and dependencies. Instructions must wait for data to propagate through the register file.	Results pass from reservation stations to functional units over a Common Data Bus that broadcasts results to all FUs. This broadcast mechanism reduces data dependencies and allows for parallel execution of independent instructions.
Load/Store Units	Treated separately, leading to potential stalls and dependencies between load/store and other instructions. Load/store instructions may wait for data to be fetched from memory.	Load and Stores treated as FUs with reservation stations, enabling them to participate in the out-of-order execution. This integration improves memory access efficiency and reduces stalls related to data dependencies.
Instruction Flow	Integer instructions wait for branches to resolve before execution, potentially causing stalls and reducing performance.	Integer instructions can proceed past branches, allowing for speculative execution and improved performance. This feature enables the algorithm to exploit instruction-level parallelism and enhance overall processor efficiency.

This detailed comparison provides an in-depth analysis of the differences between the Scoreboard and Tomasulo Algorithm in terms of control logic, buffers, register handling, optimizations, data flow, load/store units, and instruction flow.

Feature	Tomasulo (IBM)	Scoreboard (CDC)
Issue Window Size	14	5
Structural Hazard Handling	No issues due to reservation stations (RS)	No issue on structural hazards
Hazard Resolution	WAR and WAW avoided with renaming	Stall for WAW and WAR hazards
Result Broadcast	Broadcast results from functional units (FUs)	Results written back on registers
Control Mechanism	Control distributed on reservation stations (RS)	Control centralized through the Scoreboard
Loop Unrolling Capability	Allows loop unrolling in hardware	Not specified

This table highlights the key differences between the Tomasulo algorithm (as implemented by IBM) and the Scoreboard method (used by CDC), focusing on their respective characteristics related to issue window size, hazard handling, result handling, control mechanism, and additional capabilities like loop unrolling.

PDF 12 - Hardware Based Speculation

HW-based Speculation combines three core ideas:

1. **Dynamic Branch Prediction**:
   - **Purpose**: Predicts the direction of branches (e.g., conditional statements) to decide which instructions to fetch and execute.
   - **Mechanisms**: Utilizes history-based algorithms and heuristics to make accurate predictions, reducing the number of pipeline stalls due to branch instructions.
2. **Dynamic Scheduling**:
   - **Purpose**: Allows instructions to be executed out of their original program order based on the availability of operands and execution units.
   - **Mechanisms**: Implements techniques such as Tomasulo's algorithm or reservation stations to track data dependencies and resource availability, enabling better utilization of the CPU's execution units.
3. **Speculative Execution**:
   - **Purpose**: Executes instructions ahead of time based on predictions, with the intention of committing the results if the predictions are correct.
   - **Mechanisms**: Uses mechanisms like reorder buffers and checkpoints to ensure that speculative results are only committed if the prediction was accurate. If the prediction was wrong, the speculative execution is rolled back.

The key ideas behind speculation are:

- **Issuing and executing instructions dependent on a branch before the branch outcome is known**:
  - This reduces idle cycles in the pipeline by allowing the processor to continue working on subsequent instructions while waiting for the branch decision.
- **Allowing instructions to execute out-of-order but forcing them to commit in-order**:
  - This ensures that the final state of the processor and memory remains consistent with the program's intended sequence, maintaining correct program behavior.
- **Preventing any irrevocable action (such as updating state or taking an exception) until an instruction commits**:
  - This safeguard ensures that any changes made speculatively do not affect the system's state permanently unless the speculative execution is confirmed to be correct, allowing for safe rollbacks if predictions are wrong.

By integrating dynamic branch prediction, dynamic scheduling, and speculative execution with these key ideas, modern processors can significantly enhance performance by maximizing instruction throughput and minimizing pipeline stalls.

Mechanisms are necessary to handle incorrect speculation–hardware speculation extends dynamic scheduling beyond a branch (i.e. behind the basic block)

The Reorder Buffer (ROB) is a critical component in modern processors, particularly in those that use out-of-order execution and speculative execution. Its primary function is to maintain the correct program order for instruction commitment, ensuring that the processor state is updated correctly even when instructions are executed out of order.

Reorder Buffer (ROB):

Purpose of the Reorder Buffer

The ROB is used to: - **Ensure In-order Commitment**: While instructions can be executed out-of-order for performance reasons, they must be committed (i.e., their results must be written to the architectural state) in the original program order. The ROB ensures this in-order commitment. - **Support Speculative Execution**: When instructions are executed speculatively, the ROB helps manage the results. If a branch prediction turns out to be incorrect, the ROB allows the processor to discard the speculative results. - **Handle Exceptions and Interrupts**: The ROB can manage exceptions and interrupts in a way that maintains program correctness by ensuring that only committed instructions affect the state.

Structure of the Reorder Buffer

The ROB is typically a circular buffer, with each entry in the buffer corresponding to a single instruction. Each entry usually contains: - **Instruction Identifier**: A unique identifier for the instruction. - **Destination Register**: The architectural register or memory location where the result should be written. - **Result Value**: The result produced by the instruction. - **Completion Status**: A flag indicating whether the instruction has completed execution. - **Exception Status**: Information on whether the instruction caused an exception.

Operation of the Reorder Buffer

1. **Instruction Issue**:
   - When an instruction is issued, an entry is allocated in the ROB. This entry holds information about the instruction, such as its destination register.
2. **Instruction Execution**:
   - As the instruction executes, it produces results. These results are written to the ROB entry rather than directly to the register file or memory.
3. **Instruction Completion**:
   - Once the instruction completes execution, its status in the ROB is updated to indicate that it has completed, and the result is available.
4. **Instruction Commitment**:
   - Instructions are committed in the order they were issued. The commit point in the ROB advances sequentially, ensuring that instructions update the architectural state (register file and memory) in program order.
   - If the instruction at the head of the ROB is complete and has no exceptions, its result is written to the destination register or memory location, and the entry is retired from the ROB.
   - If an instruction causes an exception, the processor can handle it appropriately without committing any subsequent instructions, maintaining a consistent state.

Commit: 3 Different Possible Sequences

1. **Normal Commit**:
   - Instruction reaches the head of the ROB, result is present in the buffer.

- Result is stored in the register.
- Instruction is removed from ROB.

2. **Store Commit**:
   - As above, but memory rather than register is updated.
3. **Instruction is a Branch with Incorrect Prediction**:
   - It indicates that speculation was wrong.
   - ROB is flushed ("graduation").
   - Execution restarts at the correct successor of the branch.
   - If the branch was correctly predicted, the branch is finished.

Handling Speculative Execution

- If a branch prediction turns out to be incorrect, the ROB allows the processor to discard all speculative instructions following the mispredicted branch. This involves invalidating the relevant ROB entries and rolling back the state to the last known good state.
- Checkpoints may be used to facilitate this rollback, making it efficient to restore the processor state.

Example

Consider a processor that has issued three instructions: 1. **Instruction 1**: Load value into register R1. 2. **Instruction 2**: Add values in registers R1 and R2, storing the result in R3. 3. **Instruction 3**: Conditional branch based on the value in R3.

- The ROB will allocate entries for each of these instructions as they are issued.
- If Instruction 3 (the branch) is executed and predicts the branch taken path, subsequent instructions will execute speculatively.
- The results of these instructions will be placed in the ROB.
- If the branch prediction was correct, the speculative results are committed in order.
- If the branch prediction was incorrect, the speculative results are discarded, and execution resumes from the correct branch target.

Benefits

- **Performance**: By allowing out-of-order execution, the processor can better utilize its execution units, improving performance.
- **Correctness**: The in-order commitment ensures that the final program state is correct, despite out-of-order execution.
- **Flexibility**: The ROB helps manage speculative execution, making modern techniques like branch prediction more effective.

In summary, the Reorder Buffer is essential for balancing the need for high performance through out-of-order and speculative execution with the requirement for correct program behavior through in-order commitment.

Separating Completion from Commit

- **Re-order buffer holds register results from completion until commit**
  - Entries allocated in program order during decode
  - Buffers completed values and exception state until in-order commit point
  - Completed values can be used by dependents before committed (bypassing)
  - Each entry holds program counter, instruction type, destination register specifier and value if any, and exception status (info often compressed to save hardware)
- **Memory reordering needs special data structures**
  - Speculative store address and data buffers
  - Speculative load address and data buffers

Each Entry in ROB Contains Four Fields:

- **Instruction Type Field**:
  - Indicates whether the instruction is a branch (no destination result), a store (has memory address destination), or a load/ALU (register destination).
- **Destination Field**:
  - Supplies register number (for loads and ALU instructions) or memory address (for stores) where results should be written.
- **Value Field**:
  - Used to hold value of result until the instruction commits.
- **Ready Field**:
  - Indicates that the instruction has completed execution, and the value is ready.

PDF 13 - ILP

Explicit Register Renaming

- **Tomasulo provides Implicit Register Renaming**:
  - User registers renamed to reservation station tags
- **Now we introduce Explicit Register Renaming**:
  - Use physical register file that is larger than the number of registers specified by the ISA
  - **Key insight**: Allocate a new physical destination register for every instruction that writes
    * Very similar to a compiler transformation called Static Single Assignment (SSA) form—but in hardware!
    * Removes all chance of WAR (Write After Read) or WAW (Write After Write) hazards
    * Like Tomasulo, good for allowing full out-of-order completion
    * Like hardware-based dynamic compilation?

```
Explicit register renaming and hardware-based dynamic compilation serve different purpose

- **Explicit Register Renaming** allocates additional physical registers to eliminate haz

- **Hardware-based Dynamic Compilation** optimizes code at runtime, using techniques like

While both techniques aim to enhance processor efficiency, explicit register renaming foc
```

Explicit Register Renaming Mechanism:

- Keep a translation table:
  - ISA register → physical register mapping
- When a register is written to, replace its entry with a new register from a free list
- Physical registers become free when not used by any active instructions

Unified Physical Register File

- **Mechanism**:
  - Rename all architectural registers into a single physical register file during decode, without reading register values.
  - Functional units read and write from a single unified register file that holds committed and temporary registers during execution.
  - Commit only updates the mapping of architectural registers to physical registers; no actual data movement occurs during commit.

Explanation:

- **Unified Physical Register File**: Instead of maintaining separate sets of registers for architectural (visible to the programmer) and physical (used internally for execution), this approach merges them into a single unified register file.
- **During Decode**: When instructions are decoded, architectural registers are mapped to specific physical registers. This mapping allows instructions to proceed with their execution using these physical registers.
- **Execution Phase**: Functional units (ALUs, load/store units, etc.) perform operations directly on the unified register file. They read operands from and write results back to this unified file, whether the data corresponds to committed (architectural) registers or temporary registers used internally by the processor.
- **Commit Phase**: When instructions complete execution and are ready to commit their results, only the mapping of architectural registers to physical registers is updated. This ensures that the architectural state of the processor accurately reflects the committed results without physically moving large amounts of data.

This approach reduces complexity in managing register states within the processor, supports efficient out-of-order execution, and simplifies the handling of architectural state updates during the commit phase.

HW Register Renaming

- **Renaming Map**:
  - A simple data structure that provides the physical register number corresponding to the requested architectural register.
- **Instruction Commit**:
  - Updates the renaming table permanently to indicate that the physical register holding the destination value now corresponds to the actual architectural register.
- **Use of ROB**:
  - Utilizes the Reorder Buffer (ROB) to enforce in-order commit, ensuring that instructions are committed in the sequence they were issued.

This setup ensures efficient management of register dependencies and supports out-of-order execution while maintaining the integrity of architectural state updates during the commit phase.

Stages of Scoreboard Control With Explicit Renaming

Execution Pipeline Overview

- **Issue**:
  - Decode instructions & check for structural hazards.
  - Allocate new physical register for result.
  - Instructions issued in program order (for hazard checking).
  - Don't issue if no free physical registers.
  - Don't issue if there is a structural hazard.
- **Read Operands**:
  - Wait until no hazards; read operands.
  - All real dependencies (RAW hazards) resolved in this stage, as we wait for instructions to write back data.
- **Execution**:
  - Operate on operands.
  - The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard.
- **Write Result**:
  - Finish execution.
- **Note**: No checks for WAR (Write After Read) or WAW (Write After Write) hazards in this overview.

This structured approach outlines the stages in the execution pipeline, emphasizing hazard checking, operand handling, execution, and result writing, crucial for managing the flow of instructions and ensuring correct and efficient processing in advanced computer architectures.

> register renaming typically eliminates Write After Write (WAW) and Write After Read (WAR) hazards in modern processors when combined with appropriate hazard detection mechanisms like scoreboarding.

Register Renaming vs. ROB

- **Instruction Commit**:
  - Simpler than with ROB.
- **Deallocating Registers**:
  - More complex.
- **Dynamic Mapping**:
  - Dynamic mapping of architectural to physical registers complicates design and debugging.
- **Usage**:
  - Used in:
    - PowerPC 603/604
    - Pentium II-III-IV
    - MIPS 10000/12000
    - Alpha 21264
    - Sandy Bridge (20 to 80 registers added)

Certainly! Here's the formatted text:

Summary

- **More Physical Registers**:
  - More physical registers than needed by the ISA.
- **Rename Table**:
  - Tracks current association between architectural registers and physical registers.
  - Uses a translation table to perform compiler-like transformations on the fly.
- **With Explicit Renaming**:
  - All registers concentrated in a single register file.
  - Can utilize a bypass network resembling a 5-stage pipeline.
  - Introduces a register allocation problem.
- **Handling Branch Misprediction and Precise Exceptions**:
  - Requires different handling, but ultimately simplifies operations.
- **For Precise Exceptions and Branch Prediction**:
  - Clearly requires a component like a reorder buffer.

This structure outlines how explicit renaming enhances processor performance by efficiently managing register allocation and simplifying complex operations like branch prediction and exception handling.

PDF 14 - Instruction Level Parallelism: Limits

Limits to ILP

Assumptions for an ideal/perfect machine to start:

1. **Register renaming**
   - Infinite virtual registers, avoiding all WAW (Write After Write) and WAR (Write After Read) hazards.
2. **Branch prediction**
   - Perfect, with no mispredictions.

3. **Jump prediction**
   - All jumps perfectly predicted, resulting in a machine with perfect speculation and an unbounded buffer of instructions available.
4. **Memory-address alias analysis**
   - Addresses are known, and a store can be moved before a load provided the addresses are not equal.
5. **1 cycle latency for all instructions**
   - An unlimited number of instructions issued per clock cycle.

Initial Assumptions

- **Unlimited instruction issue**
  - The CPU can issue an unlimited number of instructions at once, looking arbitrarily far ahead in computation.
- **No instruction type restrictions**
  - There are no restrictions on the types of instructions that can be executed in one cycle, including loads and stores.
- **Functional unit latencies**
  - All functional unit latencies are 1 cycle, allowing any sequence of dependent instructions to issue on successive cycles.
- **Perfect caches**
  - All loads and stores execute in one cycle, meaning only the fundamental limits to ILP are taken into account.
- **Note**
  - The results obtained under these assumptions are VERY optimistic, as no such CPU can be realized.
- **Benchmark programs**
  - Six programs from SPEC92 were used, including three FP-intensive ones and three integer ones.

Limits on Window Size

- **Dynamic analysis necessity**
  - Dynamic analysis is essential to approach perfect branch prediction, which is impossible to achieve at compile time.
- **Requirements for a perfect dynamically-scheduled CPU:**
  1. **Instruction lookahead and branch prediction**
     - Look arbitrarily far ahead to find a set of instructions to issue and predict all branches perfectly.
  2. **Register renaming**
     - Rename all register uses to avoid WAW (Write After Write) and WAR (Write After Read) hazards.
  3. **Data dependency determination**
     - Determine whether there are data dependencies among instructions in the issue packet and rename registers if necessary.
  4. **Memory dependency handling**
     - Determine if memory dependencies exist among issuing instructions and handle them accordingly.
  5. **Functional unit replication**
     - Provide enough replicated functional units to allow all ready instructions to issue.

Limits on Instruction Windows

- **Impact of Window Size**

  - The size of the instruction window affects the number of comparisons necessary to determine Read After Write (RAW) dependencies.
- **Example: Comparisons for Data Dependencies**
  - To evaluate data dependencies among (n) register-to-register instructions in the issue phase (with an infinite number of registers): Number of Comparisons = (n * (n - 1))
    - **Window size = 2000**
      - This requires almost 4 million comparisons!
    - **Issue window of 50 instructions**
      - This requires 2450 comparisons!
- **Constraints in Today's CPUs**
  - The limited number of registers.
  - The need to search for dependent instructions.
  - The necessity to issue instructions in order.

Limits on Window Size and Maximum Issue Count

- **Instruction Retention**
  - All instructions in the window must be stored within the processor.
- **Comparison Requirements**
  - The number of comparisons required at each cycle is calculated as:
    `Maximum completion rate × Window size × Number of operands per instruction`
- **Constraints on Window Size**
  - The total window size is limited by:
    `Storage capacity. + The number of comparisons needed. + The limited issue rate.`
- **Current CPU Capabilities**
  - Modern CPUs have window sizes ranging from 32 to 200, resulting in up to over 2400 comparisons per cycle.

Other Limits of Today's CPUs

- **Number of Functional Units**
  - For example, not more than 2 memory references can be handled per cycle.
- **Number of Buses**
- **Number of Ports for the Register File**

These limitations mean that the maximum number of instructions that can be issued, executed, or committed in the same clock cycle is much smaller than the window size.

Current Superscalar & VLIW processors

- Dynamically-scheduled superscalar processors are the commercial state-of-the-art for general purpose: current implementations of Intel Core i , PowerPC, Alpha, MIPS, SPARC, etc. are all superscalar
- VLIW processors are primarily successful as embedded media processors for consumer electronic devices

Taxonomy of Multiple Issue Machines

Here's the updated table without SIMD and including the three types of Superscalar architectures:

Common Name	Issue Structure	Hazard Detection	Scheduling	Distinguishing Characteristics	Examples
Superscalar (Static)	Multiple instructions per cycle	Dynamic (at runtime)	In-order	Issues multiple instructions per cycle but executes them in program order; simpler pipeline design compared to out-of-order superscalar	Intel Pentium, ARM Cortex-A5
Superscalar (Dynamic)	Multiple instructions per cycle	Dynamic (at runtime)	Out-of-order	Issues and executes instructions out of program order to maximize pipeline utilization and performance; sophisticated hazard detection and handling	Intel Core i7, AMD Ryzen
Superscalar (Speculative)	Multiple instructions per cycle	Dynamic (at runtime)	Speculative	Executes instructions speculatively based on predictions to achieve higher performance; handles mispredictions carefully to maintain correctness	Intel Pentium Pro, ARM Cortex-A76
VLIW (Very Long Instruction Word)	Single instruction with multiple operations	Static (at compile time)	Parallel (by compiler)	Combines multiple operations into a single instruction word; requires compiler to schedule operations and ensure no dependencies	Intel Itanium, TI TMS320C6x
EPIC (Explicitly Parallel Instruction Computing)	Multiple independent instructions issued together	Static (compiler determines)	Parallel (by compiler)	Employs a wide-issue architecture; relies on compiler to identify and schedule independent instructions for simultaneous execution	Intel IA-64 (Itanium), HP PA-8000

Limits to ILP

Increasing processor performance typically raises power consumption due to the overhead incurred by most techniques. The critical consideration is whether a technique is energy-efficient—does it boost performance faster than it escalates power use?

Multiple issue processors exemplify energy inefficiency: - Issuing multiple instructions involves increasing logic overhead that outpaces the growth in issue rates. - This results in a widening gap between peak issue rates and sustained performance. - The number of transistors switching increases with the peak issue rate, while performance correlates with the sustained rate, further widening the energy per unit of performance.

For instance, the Itanium 2, despite its wide issue capabilities, operates at a slower clock rate and consumes more power, highlighting the trade-offs between performance enhancement and energy efficiency in modern processors.

Parallel Architectures

- **Definition:**
  - "A parallel computer is a collection of processing elements that cooperates and communicates to solve large problems fast."
  - *(Almasi and Gottlieb, Highly Parallel Computing, 1989)*

- The goal is to replicate processors to enhance performance rather than designing a faster single processor.
- Parallel architecture extends traditional computer architecture with a **communication architecture**:
  - Abstractions for hardware/software interfaces.
  - Utilizes various structures to efficiently realize these abstractions.

the four types in Flynn's taxonomy of parallel computer architectures:

Category	Description	Characteristic	Examples
SISD (Single Instruction, Single Data)	Traditional von Neumann architecture where a single instruction operates on a single data stream.	Sequential execution of instructions on a single processor.	Classic CPUs (e.g., early computers like ENIAC, modern desktop/laptop processors).
SIMD (Single Instruction, Multiple Data)	Executes the same instruction simultaneously on multiple data points.	Parallel execution with a single control unit issuing instructions to multiple processing elements.	GPU shaders, SIMD extensions in CPUs (e.g., Intel SSE, ARM NEON).
MISD (Multiple Instruction, Single Data)	Rarely implemented; multiple instructions operate on a single data stream.	Concurrency achieved through different processing units applying different operations to the same data.	Hypothetical or experimental architectures (not commonly found in practical systems).
MIMD (Multiple Instruction, Multiple Data)	Multiple processors executing different instructions on different data streams concurrently.	Highest level of parallelism with multiple independent processors working on multiple sets of data.	Clusters of workstations, modern multi-core processors, distributed computing systems (e.g., HPC clusters).

This table summarizes Flynn's taxonomy, distinguishing each category by its operational characteristics, mode of instruction/data processing, and examples of real-world implementations or theoretical constructs.

PDF 15 - Multiprocessors

Many of the early multiprocessors were SIMD

MIMD has emerged as architecture of choice for general-purpose multiprocessors

SIMD Architecture:

Central controller broadcasts instructions to multiple processing elements (PEs)

- Only requires one controller for whole array
- Only requires storage for one copy of program
- All computations fully synchronized

SIMD model

- **Synchronized units: single Program Counter**
- Each unit has its own addressing registers, allowing different data addresses.
- **Motivations for SIMD:**
  - Cost efficiency due to shared control unit across all execution units.
  - Only one instance of code execution required.
- **Real-life implementation:**
  - SIMD architectures often mix SISD (Single Instruction, Single Data) and SIMD instructions.
  - Host computers execute sequential operations alongside SIMD instructions.
  - Execution units within SIMD have their own memory and registers, utilizing an interconnection network for data exchange. ## Alternative Model: Vector Processing

Vector processors have high level operations that work on linear arrays of numbers: "vectors"

Styles of Vector Architectures

- **Vector Processor Components:**
  - A vector processor combines a pipelined scalar unit (which may be out-of-order or VLIW) with a vector unit.
- **Memory-memory vector processors:**
  - All vector operations involve memory-to-memory interactions.
- **Vector-register processors:**
  - Vector operations occur between vector registers, except for load and store operations.
  - This architecture is akin to load-store architectures but optimized for vectors.
  - Implemented in vector machines since the late 1980s, including Cray, Convex, Fujitsu, Hitachi, NEC.

Vector Arithmetic Execution

Vector arithmetic execution utilizes a deep pipeline to achieve fast clock speeds. Here's a clearer explanation of the key points:

- **Deep Pipeline**: Vector processors employ a deep pipeline structure, allowing them to operate at high clock frequencies. A deep pipeline means that the execution of instructions is divided into several stages, with each stage handling a specific part of the instruction execution process.
- **Simplified Control**: The independence of elements within a vector simplifies pipeline control. In a vector operation, each element of the vector operates independently of others. This independence means there are no dependencies or hazards between elements that would stall the pipeline. Therefore, the control of the deep pipeline is streamlined because there's no need for complex hazard detection and handling mechanisms.

In essence, vector processors leverage a deep pipeline and the independence of vector elements to achieve high performance by executing operations swiftly and efficiently.

Vector Applications Beyond Scientific Computing

- **Multimedia Processing**: Includes compression (JPEG, MPEG), graphics rendering, audio synthesis, and image processing.
- **Standard Benchmark Kernels**: Such as Matrix Multiply, FFT (Fast Fourier Transform), Convolution, and Sort algorithms.
- **Compression Techniques**: Both lossy (JPEG, MPEG for video and audio) and lossless methods (Zero removal, Run-Length Encoding (RLE), Lempel-Ziv-Welch (LZW)).
- **Cryptography**: Used in RSA encryption, DES/IDEA encryption algorithms, and hashing algorithms like SHA and MD5.
- **Speech and Handwriting Recognition**: Processing and analyzing large datasets of audio and handwritten inputs.
- **Operating Systems/Networking**: Utilized for operations like memory copying (memcpy), memory setting (memset), parity calculation, and checksum generation.
- **Databases**: Employed for tasks such as hashing and joining datasets, data mining, and serving image and video data.
- **Language Runtime Support**: Providing standard library functions and managing memory through garbage collection.
- **Performance Benchmarking**: Used in benchmarks like SPECint95 to measure and compare computational performance across different platforms.

  MIMDs are flexible. they can function as single-user machines for high performances on one application, as multiprogrammed multiprocessors running many tasks simultaneously, or as some combination of such functions;

  Can be built starting from standard CPUs (such is the present case nearly for all multiprocessors!).

MIMD - Multiple Instruction Multiple Data

- Each processor fetches its own instructions and operates on its own data.
- Processors are often off-the-shelf microprocessors.
- Scalable to a variable number of processor nodes.
- **Flexible:**
  - Single-user machines optimized for high-performance in specific applications.
  - Multi-programmed machines capable of running many tasks simultaneously.
  - Combination of these functions.
- Cost/performance advantages due to the use of off-the-shelf microprocessors.
- **Fault Tolerance Issues:**
  - Ensuring reliability and continuity of operations despite hardware failures.
- To exploit MIMD with (n) processors:
  - Requires at least (n) threads or processes for execution.
  - Threads are typically identified by programmers or created by compilers.
  - Parallelism is managed at the thread level, known as **thread-level parallelism**.
- **Thread**: Can range from a large, independent process to parallel iterations of a loop. Note: Parallelism is identified and managed by software, not hardware as in superscalar CPUs.

MIMD Machines

MIMD machines are categorized into two classes based on the number of processors involved, influencing memory organization and interconnection strategies:

Centralized Shared-Memory Architectures

- **Processor Count:** Typically involves a few dozen processor chips (< 100 cores).
- **Memory Organization:**
  - Utilizes large caches.
  - Implements single memory with multiple banks.
- **Architecture Style:** Often referred to as Symmetric Multiprocessors (SMP) with Uniform Memory Access (UMA).

Distributed Memory Architectures

- **Support for Large Processor Counts:**
  - Requires high-bandwidth interconnect.
- **Disadvantages:**
  - Involves challenges in data communication among processors due to the distributed memory setup.

These architectures vary significantly in how they handle memory access and interconnectivity, impacting their scalability and performance in parallel computing applications.

I skipped game consoles and microsoft's vision

From ILP to TLP: From the Processor to the Programmer

Transitioning from ILP (Instruction-Level Parallelism) to TLP (Thread-Level Parallelism) simplifies processor design by removing hardware intended for optimizing instruction scheduling at runtime. Here's a clearer explanation of the key points:

- **Simplified Design Approach:**
  - Hardware optimizing instruction scheduling at runtime is stripped out.
  - Processors like Xenon and Cell do not incorporate an instruction window.
- **Order of Execution:**
  - Instructions are processed in the order they are fetched.
  - Adjacent, non-dependent instructions are executed in parallel whenever possible.
- **Static Execution:**
  - Implementation is straightforward.
  - Requires significantly less die space compared to dynamic execution.
- **Die Space Efficiency:**
  - Without an instruction window, more transistors are available for additional execution units.
  - This optimizes the die space usage by focusing on increasing actual execution capabilities rather than complex scheduling hardware.
- **Rethinking Processor Organization:**
  - Eliminating the instruction window cannot simply be replaced by adding more execution units.
  - Requires a reconsideration of how the processor's resources are organized and utilized to effectively handle parallelism at the thread level.

This approach shifts the focus from intricate runtime optimizations towards a more efficient use of hardware resources to enhance thread-level parallelism and overall processor performance.

Procedural Synthesis in a Nutshell

Procedural synthesis is a technique used to enhance the efficiency of rendering in computer graphics. It achieves this by dynamically generating detailed geometry data at runtime, directly from higher-level scene descriptions that are stored statically. This approach helps optimize both system bandwidth and main memory usage.

Here's how it works:

- **For 3D Games:**
  - Artists use 3D rendering programs to create game content.
  - Each model is converted into collections of polygons.

  - These polygons are represented in the computer's memory as sets of vertices.
- **Real-time Rendering in Games:**
  - Models displayed on-screen are initially stored as vertex data in main memory.
  - This vertex data is then sent from main memory to the GPU.
  - The GPU renders this data into a 3D image, which is output to the monitor as a sequence of frames.

Limitations

However, procedural synthesis faces several challenges:

- **Increasing Costs of Art Assets:**
  - The expense of creating 3D game assets is rising due to the growing size and complexity of games.
- **Hardware Limitations:**
  - Console hardware often has limited main memory sizes and bandwidth, which can restrict the efficiency of procedural synthesis in real-time rendering scenarios.

PDF 16 - GPUs and Heterogeneous Computing Systems

Xbox 360's Approach

- **High-level Object Descriptions:**
  - Objects are stored in main memory as high-level descriptions.
- **Dynamic Geometry Generation:**
  - The CPU dynamically generates object geometry during runtime.
  - For example, vertex data is generated by running threads.
- **GPU Rendering:**
  - The GPU utilizes this dynamically generated vertex information to render objects.
  - It treats the data as if it were retrieved directly from main memory.

This approach optimizes memory usage and processing efficiency by dynamically creating detailed geometry only when needed, enhancing overall rendering performance on the Xbox 360.

GPU vs CPU

Aspect	GPU	CPU
Parallelism	Tailored for highly parallel operation	Executes programs serially
Execution Units	Many parallel execution units	Few execution units, higher clock speeds
Transistor Count	Higher transistor count	Lower transistor count
Determinism	Generally deterministic in operation (changing slowly)	Deterministic in operation
Pipeline Depth	Deeper pipelines (several thousand stages)	Shallow pipelines (10-20 stages)
Memory Interfaces	Faster and more advanced memory interfaces	Slower memory interfaces, optimized for lower data throughput

This table highlights the key differences between GPUs and CPUs, emphasizing their respective strengths in handling parallelism, execution units, transistor counts, determinism, pipeline depth, and memory interface capabilities.

The GPU pipeline

The GPU receives geometry information from the CPU as an input and provides a picture as an output

Host Interface → Vertex Processing → Triangle Setup → Pixel Processing → Memory Interface

Host Interface

- The host interface serves as the communication bridge between the CPU and the GPU.
- It receives commands from the CPU and retrieves geometry information from system memory.
- The host interface outputs a stream of vertices in object space along with associated information such as normals, texture coordinates, per-vertex color, etc.

Vertex Processing

- The vertex processing stage receives vertices from the host interface in object space and outputs them in screen space.
- This transformation can range from a simple linear transformation to complex operations involving morphing effects.
- Normals, texture coordinates, and other associated data are also transformed.
- No new vertices are created or discarded in this stage; there is a 1:1 mapping between input and output vertices.

Triangle Setup

- In this stage, geometric information is converted into raster information.
  - Screen space geometry serves as the input, and pixels are the output.
- Before rasterization, triangles that are backfacing or located outside the viewing frustum are typically rejected.
- Some GPUs also perform hidden surface removal at this stage, enhancing rendering efficiency by discarding non-visible surfaces before pixel processing.

  Rasterization is the process of converting vector-based geometric shapes or objects into a raster image composed of pixels. In computer graphics, rasterization involves determining which pixels on the screen should be filled based on the geometric properties (such as vertices and edges) of the shapes being rendered. This process is essential for displaying graphics on a digital screen, where each pixel represents a discrete point of color or intensity. Rasterization ensures that the geometric shapes defined in a 3D scene are accurately represented on a 2D display by mapping them to the appropriate pixels.

Fragment Processing

- Each fragment provided by triangle setup is fed into fragment processing as a set of attributes (position, normal, texture coordinates, etc.), which are used to compute the final color for each pixel.
- The computations in this stage include texture mapping and various mathematical operations.
- Fragment processing is typically the bottleneck in modern applications, as it involves intensive calculations to determine the final appearance of each pixel in the rendered image.

Memory Interface

- Fragment colors computed by the previous stage are written to the framebuffer.
- Historically, this stage used to be the biggest bottleneck before fragment processing became more computationally intensive.
- Before fragments are finally written, some are rejected based on tests such as z-buffer, stencil, and alpha tests.
- In modern GPUs, z and color data are often compressed to reduce framebuffer bandwidth, although this compression does not typically affect the size of the framebuffer itself.

In the graphics processing pipeline, the journey from the host interface to the memory interface involves several key stages. Initially, the host interface acts as a link between the CPU and GPU, handling commands and fetching geometry data from system memory. This data includes vertices with associated attributes like normals and texture coordinates. Vertex processing then transforms these vertices from object space to screen space, performing necessary computations without creating or discarding vertices. Next, in triangle setup, geometric data is rasterized into pixel data for rendering, where backfacing triangles and those outside the view frustum are often rejected. Following rasterization, fragment processing computes final pixel colors using attributes from vertex processing, which can include texture mapping and complex mathematical operations, often representing a performance bottleneck. Finally, in the memory interface stage, computed fragment colors are written to the framebuffer after passing through tests like z-buffering and compression to optimize framebuffer bandwidth. This comprehensive process ensures accurate rendering of 3D scenes on 2D displays with efficient use of GPU resources.

Programmability in the GPU

- Vertex, fragment processing, and now triangle setup are programmable stages.
- Programmers can write custom programs executed for each vertex and fragment.

This programmability enables the creation of highly customizable geometry and shading effects, surpassing the generic appearance of older 3D applications.

GPU/CPU Intercation

The CPU and GPU inside the system work in parallel with each other

There are two threads going on, one for the CPU and one for the GPU, which communicate through a command buffer

CPU writes commands on the buffer and GPU reads them

CPU/GPU Interaction (cont.)

- If the command buffer is drained empty, the system is CPU limited, causing the GPU to idle while waiting for new input.
  - **Note:** Despite powerful GPUs, applications won't run faster if the CPU cannot keep up.
- Conversely, if the command buffer fills up, the system becomes GPU limited, causing the CPU to wait for the GPU to process the commands. > Programs utilizing the GPU do not adhere to the traditional sequential execution model, as tasks are parallelized across multiple processing units on the GPU.

CPU Program Behavior

In the CPU program below, the object is not drawn immediately after statement A and before statement B. Instead, the API call merely adds the command to draw the object to the GPU command buffer.

```
> Statement A
> API call to draw object
> Statement B
```

his leads to a number of synchronization considerations

Synchronization Issues (cont.)

In computer graphics, synchronization between the CPU and GPU is crucial for efficient operation. Modern APIs use semaphore-style mechanisms to manage data dependencies and ensure that the CPU and GPU coordinate effectively.

When the CPU attempts to modify data that is currently referenced by a pending GPU command, it must wait until the GPU completes its processing to avoid conflicts. This waiting period, known as spinning, halts CPU execution, which is inefficient because the CPU could otherwise be performing other useful computations.

Moreover, as the GPU processes commands from the command buffer, it gradually consumes a significant portion of its capacity. This consumption reduces the GPU's ability to execute commands in parallel with the CPU, potentially leading to performance bottlenecks in applications that rely heavily on graphics processing.

Effective synchronization mechanisms and careful management of command buffer usage are essential to maximize the overall efficiency and performance of CPU-GPU interactions in graphics-intensive applications.

To avoid these problems:

Inlining Data

Inlining data refers to the practice of embedding small or frequently accessed data directly into the program code or into data structures, rather than accessing it through separate memory locations. This technique aims to improve performance by reducing memory access latency and potentially optimizing cache usage. By avoiding the overhead associated with fetching data from memory, inlined data can lead to faster execution times, especially in scenarios where data access speed is critical.

Renaming Data

Renaming data in the context of GPUs typically refers to techniques used to optimize memory access and reduce dependencies among instructions. In GPU architectures, registers are often renamed to allow multiple threads or processes to execute concurrently without interference. This helps in avoiding data hazards such as read-after-write (RAW) and write-after-read (WAR) conflicts by assigning temporary or virtual names to registers, ensuring that each thread operates on its own distinct data set. By renaming registers dynamically, GPUs can effectively increase instruction-level parallelism (ILP) and optimize the utilization of hardware resources.

GPU Readbacks

GPU readbacks involve transferring data from the GPU back to the CPU's main memory. This process is necessary when the CPU needs to access results or data processed by the GPU. While GPU computations are typically faster due to their highly parallel nature and specialized hardware, readbacks can introduce performance overheads. This is because transferring data between the GPU and CPU involves crossing the PCIe bus, which has limited bandwidth compared to internal GPU memory access. Therefore, minimizing the frequency and volume of GPU readbacks is crucial for optimizing overall system performance in GPU-accelerated applications.

Here are some GPU optimization tips:

- **Batch Dispatch:** Utilize the GPU's parallelism and pipelining capabilities by dispatching large batches of work with each drawing call. Sending small batches or individual primitives (like triangles) underutilizes the GPU's processors and pipelines.
- **Rendering Order:** When rendering objects, consider front-to-back ordering rather than back-to-front or random ordering. GPUs typically use the z-buffer algorithm for hidden surface removal, and rendering front-to-back can optimize efficiency by minimizing overdraw.
- **Consider CPU Limitations:** Front-to-back sorting is beneficial primarily when the GPU, not the CPU, is the limiting factor in rendering performance. If the CPU is already operating at its maximum capacity, optimizing rendering order may not yield significant performance improvements.

Certainly! Here's the formatted text:

A Specialized Processor

- **Very Efficient For**
  - Fast Parallel Floating Point Processing
  - Single Instruction Multiple Data Operations
  - High Computation per Memory Access
- **Not As Efficient For**
  - Double Precision
  - Logical Operations on Integer Data
  - Branching-Intensive Operations
  - Random Access, Memory-Intensive Operations

This outline succinctly highlights the strengths and weaknesses of a specialized processor.

PDF 17 - Parallel Processors:MIMD Architectures

How do parallel processors share data?

The Memory Address Space Model refers to how a computer system organizes and manages memory addresses for accessing data and instructions within its memory hierarchy. It encompasses the organization, allocation, and access mechanisms for memory locations, providing a structured way for software and hardware components to interact with memory.

Key aspects of the Memory Address Space Model include:

1. **Addressability:** Memory is divided into discrete units, typically bytes or words, each identified by a unique address. The size of these units and the range of addresses define the total addressable memory space.
2. **Addressing Modes:** Different systems employ various addressing modes to specify how addresses are computed or accessed. This can include direct addressing (explicitly specifying an address), indirect addressing (using a pointer or reference to access data indirectly), and indexed addressing (using an offset or index to compute an address).
3. **Memory Segmentation:** Some architectures organize memory into segments, where each segment represents a portion of the address space with its own base address and limit. Segmentation allows for more flexible memory management and protection mechanisms.
4. **Memory Protection:** Modern systems implement memory protection mechanisms to control access to memory regions. This ensures that programs do not interfere with each other's memory space, enhancing security and stability.
5. **Virtual Memory:** Virtual memory systems extend physical memory by using disk storage as an extension, allowing larger programs or multiple programs to run simultaneously. It provides the illusion of a larger memory space than physically available by swapping data between physical RAM and disk storage.
6. **Address Space Layout:** The layout of memory addresses can vary between systems, including the arrangement of code, data, stack, and heap segments within the address space. This layout influences program execution and memory access patterns.

Overall, the Memory Address Space Model defines the framework within which programs interact with memory, ensuring efficient and secure utilization of system resources across various computing platforms.

Memory Address Space

Single logically shared address space: A memory reference can be made by any processor to any memory location

Shared Memory Architectures. The address space is shared among processors: The same physical address on 2 processors refers to the same location in memory

Here's the formatted text:

Shared Address

- **The processors communicate among them through shared variables in memory.**
- **Implicit management of the communication through load/store operations to access any memory locations.**
  - Oldest and most popular model.
- **Shared memory does not mean that there is a single centralized memory.**

This layout organizes the information clearly, highlighting the key points about the Shared Address memory model.

Memory Address Space Model

Here's the formatted text with a visual example:

Single Logically Shared Address Space

- **Memory reference can be made by any processor to any memory location**
  - **Shared Memory Architectures**
    - The address space is shared among processors: The same physical address on 2 processors refers to the same location in memory.

Multiple and Private Address Spaces

- **Processors communicate through send/receive primitives**
  - **Message Passing Architectures**
    - The address space is logically disjoint and cannot be addressed by different processors: the same physical address on 2 processors refers to 2 different locations in 2 different memories.

Visual Example:

```
Shared Memory Architectures:
Processor 1                      Processor 2
  |                                |
  V                                V
Memory:                           Memory:
Address 0x1000: Data A           Address 0x1000: Data A

Message Passing Architectures:
Processor 1                      Processor 2
  |                                |
  V                                V
Memory of P1:                     Memory of P2:
Address 0x1000: Data A           Address 0x1000: Data B
```

In Shared Memory Architectures, both processors access the same physical memory address (0x1000) and retrieve the same data (Data A). In Message Passing Architectures, the same physical address (0x1000) on different processors refers to different locations (Data A in Processor 1's memory and Data B in Processor 2's memory).

Multiple and Private Addresses

In architectures with multiple and private addresses, processors communicate through sending and receiving messages rather than directly accessing shared memory locations. Each processor manages its own private memory space independently.

Communication between processors is explicitly managed using send and receive primitives, which allow them to exchange data via messages. Unlike shared memory architectures where multiple processors can access the same memory location, here each processor accesses its own private memory locations exclusively.

This approach avoids cache coherency issues that arise in shared memory systems, where multiple processors accessing and modifying shared data can lead to inconsistencies or conflicts. By maintaining private memory spaces, each processor retains control over its data without needing to synchronize or manage shared memory accesses with other processors.

Physical Memory Organization

Architecture Type	**Centralized Shared-Memory Architectures (SMP/UMA)**	**Distributed Memory Architectures (NUMA)**
Characteristics	- Few dozen processor chips (< 100 cores)	- Supports large processor counts
	- Large caches, multiple banks of shared memory	- Requires high-bandwidth interconnect for communication between processors
	- Uniform Memory Access (UMA), where access time to any memory location is uniform	- Non-Uniform Memory Access (NUMA), where access time to memory can vary
	- Often used in symmetric multiprocessors (SMP)	- Communication among processors is a challenge

Programming Models

Shared Memory Programming Model

Concept: - In a shared memory model, multiple processors communicate by accessing shared variables in a common memory space.

Characteristics: - **Implicit Communication:** Communication between processors is managed implicitly through load/store operations. When one processor updates a shared variable, other processors can access the updated value directly from memory. - **Shared Address Space:** All processors share a single, global address space. This means that a memory address used by one processor refers to the same location as the same address used by another processor. - **Synchronization:** Mechanisms such as locks, semaphores, and barriers are often used to coordinate access to shared data and ensure consistency. - **Cache Coherency:** Ensuring that all processors have a consistent view of memory can be challenging. Cache coherency protocols are often used to maintain consistency.

Pros: - Simplicity in data sharing: Direct access to shared variables can be simpler to program. - Flexibility: Suitable for a variety of applications, especially those requiring frequent read/write access to shared data.

Cons: - Scalability: As the number of processors increases, the complexity of maintaining cache coherency and synchronization increases. - Performance: Contention for shared memory and synchronization overhead can impact performance.

Example Use Cases: - Multi-threaded applications on a single machine. - High-performance computing tasks where shared data structures are frequently accessed.

Message Passing Programming Model

Concept: - In a message passing model, processors communicate by explicitly sending and receiving messages. Each processor has its own private memory space.

Characteristics: - **Explicit Communication:** Communication is explicitly managed through send/receive operations. Processors must send messages to each other to exchange data. - **Private Address Spaces:** Each processor has its own separate address space. A memory address in one processor's space does not refer to the same location as the same address in another processor's space. - **Synchronization:** The act of sending and receiving messages inherently provides synchronization. However, additional synchronization mechanisms may be used if needed. - **No Cache Coherency Issues:** Since each processor has its own private memory, there are no cache coherency problems.

Pros: - Scalability: Easier to scale to a large number of processors since there is no need for maintaining a consistent global memory. - Performance: Avoids contention for shared memory, and communication can be optimized for the specific network topology.

Cons: - Complexity: Programming can be more complex due to the need to explicitly manage communication and data distribution. - Latency: Communication latency can be higher compared to direct memory access in shared memory models.

Example Use Cases: - Distributed systems and cluster computing. - Parallel applications that run on multiple machines, such as scientific simulations and large-scale data processing.

Summary: - **Shared Memory Model**: Processors communicate through a common shared memory. Suitable for systems with a smaller number of processors and applications requiring frequent access to shared data. - **Message Passing Model**: Processors communicate by sending and receiving messages. Suitable for scalable systems with a large number of processors and distributed applications.

Understanding these two models helps in choosing the appropriate approach based on the specific requirements and constraints of the application being developed.

Which is Better? Shared Memory (SM) or Message Passing (MP)?

Advantages of Message Passing: - **Explicit Communication:** Sending and receiving of messages. - **Easier Control of Data Placement:** No automatic caching.

Disadvantages of Message Passing: - **High Overhead:** Message passing overhead can be quite high. - **Complex Programming:** More complex to program. - **Reception Technique:** Introduces the question of reception technique (interrupts/polling).

Message Passing in Massively Parallel Processors: Problems

- **Data Layout Management:**
  - All data layout must be handled by software.
  - Cannot retrieve remote data except with message request/reply.
- **High Software Overhead:**
  - Early machines had to invoke the OS on each message (100 µs - 1 ms/message).
  - Even user-level access to the network interface incurs dozens of cycles overhead (network interface might be on the I/O bus).
  - Sending messages can be relatively cheap (similar to stores).
  - Receiving messages is expensive, requiring polling or interrupts.

Bus-Based Symmetric Shared Memory

- **Dominance and Market Presence:**
  - Dominate the server market even now.
  - Building blocks for larger systems; arriving to desktop.
- **Attractiveness:**

  - Attractive as throughput servers and for parallel programs.
  - Fine-grain resource sharing.
  - Uniform access via loads/stores.
  - Automatic data movement and coherent replication in caches.
  - Cheap and powerful extension.
- **Mechanisms:**
  - Normal uniprocessor mechanisms to access data.
  - Key is the extension of memory hierarchy to support multiple processors.

Shared Memory Machines

- **Categories:**
  - Non-cache coherent.
  - Hardware cache coherent.
- **Performance and Flexibility:**
  - Will work with any data placement (but might be slow).
  - Can choose to optimize only critical portions of code.
- **Communication:**
  - Load and store instructions used to communicate data.
  - No OS involvement.
  - Low software overhead.
- **Special Features:**
  - Usually some special synchronization primitives.
- **Large Scale Systems:**
  - Logically distributed shared memory is implemented as physically distributed memory modules.

The Problem of Cache Coherence

In shared-memory architectures, both private data (used by a single processor) and shared data (used by multiple processors) are cached. This caching introduces several issues, especially when it involves shared data.

Key Points:

- **Private and Shared Data Caching:**
  - **Private Data:** Used by a single processor.
  - **Shared Data:** Used by multiple processors to facilitate communication.
- **Replication of Shared Data:**
  - Shared values can be replicated in multiple caches.
  - Benefits include:
    - Reduced access latency.
    - Decreased required memory bandwidth.
    - Lower contention when multiple processors read the shared data simultaneously.
- **Cache Coherence Problem:**
  - Private processor caches can hold copies of the same variable.
  - A write operation by one processor may not be immediately visible to other processors.
  - Multiple copies of the same data across different caches lead to **cache coherence** issues.

Explanation: Cache coherence refers to the challenge of ensuring that all copies of a shared variable in different caches reflect the most recent update. Without coherence protocols, inconsistencies can arise, causing processors to work with stale data, leading to incorrect program execution. This problem becomes critical as more processors and cores are integrated into shared-memory systems.

What Does Coherency Mean?

Informal Definition:

- **Strict Definition:**
  - "Any read must return the most recent write."
  - This is too strict and difficult to implement.

Practical Definition:

- **Better Definition:**
  - "Any write must eventually be seen by a read."
  - All writes must be seen in the proper order, known as "serialization."

Ensuring Coherency: To ensure coherency, two main rules must be followed:

1. **Visibility of Writes:**
   - If Processor P writes to location x and Processor P1 reads from x, P's write will be seen by P1 if:
     - The read and write are sufficiently far apart.
     - No other writes to x occur between these two accesses.
2. **Serialization of Writes:**
   - Writes to a single location are serialized:
     - Two writes to the same location by any two processors are seen in the same order by all processors.
     - The latest write will always be seen.
     - This prevents illogical ordering, where an older value could be seen after a newer value.

These rules ensure that all processors have a consistent view of memory and that the most recent updates are reflected correctly across the system.

Potential Solutions: Cache Coherence Protocols

To maintain coherency in multiprocessor systems, hardware-based solutions, known as cache-coherence protocols, are employed. The key issue in implementing these protocols is tracking the status of any sharing of a data block.

There are two primary classes of protocols:

Snooping Protocols

- **Mechanism:**
  - All caches monitor (or "snoop") on a common communication medium, such as a bus, to track data.
  - When a processor wants to read or write to a memory location, it broadcasts the request on the bus.
  - All other caches check if they have a copy of the data and take appropriate action to ensure coherence.
- **Advantages:**
  - Simplicity and ease of implementation.
  - Effective for systems with a small number of processors.
- **Disadvantages:**
  - Limited scalability due to the need for all caches to monitor the common bus.
  - High bus traffic, which can become a bottleneck in larger systems.

Directory-Based Protocols

- **Mechanism:**

  - Uses a directory to keep track of the sharing status of data blocks.
  - The directory maintains information about which caches have copies of each memory block.
  - When a processor wants to read or write a memory location, it queries the directory to determine the current status of the data.
  - The directory coordinates the actions needed to maintain coherence.
- **Advantages:**
  - Better scalability than snooping protocols.
  - Reduced bus traffic since not all caches need to monitor all transactions.
- **Disadvantages:**
  - More complex to implement due to the need for a centralized directory.
  - Potential bottlenecks if the directory itself becomes overloaded.

These protocols ensure that all processors in a multiprocessor system have a consistent view of memory, avoiding issues where multiple copies of the same data might lead to incoherent states. By tracking the status of shared data, these protocols maintain data integrity and coherence across the system.

Two Basic Snooping Protocols

Snooping protocols are fundamental in maintaining cache coherence in multiprocessor systems. There are two primary types of snooping protocols: **write-invalidate** and **write-update**. Each protocol has its own mechanism for ensuring that all caches have a consistent view of the memory.

Write-Invalidate Protocol

- **Mechanism:**
  - When a processor wants to write to a cache block, it sends an invalidate signal on the bus to all other caches.
  - All other caches invalidate their copy of the block.
  - After invalidation, the writing processor can update the block in its cache, ensuring it has the only valid copy.
  - Subsequent reads by other processors will miss in their caches and will have to fetch the updated block from the writing processor or memory.
- **Advantages:**
  - Ensures that only one processor has the right to write to a block, avoiding conflicts.
  - Reduces the amount of data transferred over the bus since only invalidation signals are broadcast.
- **Disadvantages:**
  - Other processors must fetch the block from memory or the writing processor after it has been invalidated, which can increase latency.

Write-Update Protocol

- **Mechanism:**
  - When a processor wants to write to a cache block, it sends an update signal on the bus with the new data.
  - All other caches that have a copy of the block update their copies with the new data.
  - This ensures that all caches have the most recent data immediately after the write operation.
- **Advantages:**
  - Reduces the need for caches to fetch updated data from memory or the writing processor since they receive updates directly.
  - Can be more efficient in systems where data is frequently read by multiple processors after being written.
- **Disadvantages:**
  - Increases bus traffic since every write operation results in an update broadcast.
  - Can be less efficient if the data is frequently written by multiple processors, leading to constant updates.

Summary

Both protocols aim to ensure that all processors have a consistent view of memory, but they differ in how they manage write operations:

- **Write-Invalidate Protocol**: Invalidation signals are used to ensure exclusive write access, reducing bus traffic but potentially increasing read latency.
- **Write-Update Protocol**: Update signals broadcast new data to all caches, reducing read latency but increasing bus traffic.

Choosing between these protocols depends on the specific characteristics and requirements of the system, such as the number of processors and the typical access patterns to shared data.

Write-Back Cache and Snooping Protocols

Concept: When a cache miss occurs, identifying the most recent data value of a cache block can be challenging because the most recent value might be in a cache rather than in memory.

Solution: Snooping Scheme

1. **Cache Misses and Writes:**
   - The same snooping scheme used for managing cache coherence during writes can be employed to handle cache misses.
2. **Snooping Mechanism:**
   - Each processor continuously monitors (or "snoops") the addresses placed on the bus.
   - If a processor detects a read request for a cache block it holds a dirty (modified) copy of, it will respond by providing the requested block.
   - The memory access that initiated the read request is aborted since the most recent data is supplied by the snooping processor.

Benefits:

- **Up-to-Date Data:**
  - Ensures that the most recent data is always retrieved, even if it resides in another processor's cache.
- **Efficiency:**
  - Reduces unnecessary memory accesses by retrieving data directly from caches, which can be faster than accessing main memory.

Snooping Protocols: An Example

Write-Invalidate Protocol, Write-Back Cache

1. **Memory Block States:**
   - **Clean (Shared):** Clean in all caches and up-to-date in memory.
   - **Dirty (Exclusive):** Dirty in exactly one cache.
   - **Not in any caches:** The block is not present in any cache.
2. **Cache Block States:**
   - **Clean (Shared/Read Only):** The block is clean, not modified, and can be read.
   - **Dirty (Modified/Exclusive):** The cache has the only copy, it is writable, and dirty (cannot be shared).
   - **Invalid:** The block contains no valid data.

Example:

- **Scenario:**
  - Processor P1 has modified a cache block and holds the dirty copy.

  - Processor P2 requests the same cache block, resulting in a cache miss.
- **Process:**
  - P2's request is broadcast on the bus.
  - P1 detects the request, identifies it has the most recent (dirty) copy, and sends the data to P2.
  - The original memory access is aborted, and P2 receives the latest data without accessing the main memory.

Explanation:

The write-back cache with snooping ensures that when a processor needs data, it gets the latest version, whether that data is in another cache or in memory. This is achieved through a snooping mechanism where processors monitor the bus for requests. If a processor has a modified copy of the requested data (dirty copy), it supplies this data directly, aborting the original memory access. This approach maintains data consistency and enhances system performance by minimizing memory access latency.

MSI Invalidate Protocol

Overview: The MSI (Modified, Shared, Invalid) Invalidate Protocol is a cache coherence protocol used in multiprocessor systems to maintain consistency among caches sharing the same memory block.

States:

- **M (Modified):** Indicates that the cache has a modified copy of the memory block. This block is writable, and changes are not reflected in the main memory.
- **S (Shared):** Indicates that the cache has a clean, read-only copy of the memory block. It can be shared by multiple caches, and any updates are propagated to other caches.
- **I (Invalid):** Indicates that the cache does not have a valid copy of the memory block. It must obtain a copy from another cache or from main memory before any operations can be performed on it.

Protocol Details:

1. **Read Operation:**
   - A read operation obtains the memory block in the "shared" state, even if it's the only copy in the cache.
2. **Write Operation:**
   - Before performing a write operation, a cache must obtain exclusive ownership of the memory block.
   - If a cache wants to write to a block currently in the "Shared" state, it initiates a BusRdx (Bus Read Exclusive) request.
     - This request causes all other caches holding the block in the "Shared" state to invalidate (demote) their copies.
     - If the block is in the "Modified" state in another cache, that cache will flush its changes back to memory or downgrade to the "Shared" state.
     - Even if the block is already in the "Shared" state (hit in S), the cache can promote it to "Modified" (upgrade).
3. **Replacement:**
   - When replacing a cache block:
     - If a block in "Shared" state is replaced, it transitions directly to "Invalid".
     - If a block in "Modified" state is replaced, it must first write back its changes to memory before transitioning to "Invalid".

Example Scenario:

- **Scenario:**

  - Processor P1 has a modified (M) copy of a memory block.
  - Processor P2 requests the same block for reading.
- **Process:**
  - P2's read request will obtain the block in the "Shared" state from P1's cache.
  - If P2 wants to modify the block, it sends a BusRdx request to invalidate (demote) other caches holding the block in "Shared" state.
  - P1's cache, holding the block in "Modified" state, will respond by flushing its changes to memory or downgrading to "Shared".

Explanation: The MSI Invalidate Protocol ensures cache coherence by maintaining three states for each memory block across multiple caches: Modified (M), Shared (S), and Invalid (I). This protocol enables efficient read and write operations among caches. Reads are serviced with the block in "Shared" state, and writes require exclusive ownership, achieved through BusRdx requests to invalidate other caches' copies. This approach minimizes data inconsistencies and ensures that all caches see the most recent data updates in a coordinated manner.

Snoopy Coherence Protocols

Complications for the Basic MSI Protocol

- **Operations are not atomic:**
  - For example, detecting a cache miss, acquiring the bus, and receiving a response are separate steps.
  - This separation creates possibilities for deadlock and race conditions.
- **Deadlock and Race Conditions:**
  - The non-atomic nature of operations in MSI protocols can lead to situations where processors deadlock or race to access shared resources.
- **Solution to Deadlock:**
  - One approach to mitigate deadlock is for the processor sending an invalidate request to hold the bus until all other processors receive the invalidate signal.

Extensions to MSI Protocol:

- **MESI Protocol:** Adds an "Exclusive" state to indicate that a clean block exists exclusively in one cache.
  - Prevents the need to issue an invalidate operation when a write occurs.
- **Owned State:**
  - Introduces an "Owned" state to enhance cache coherence protocols, indicating that a cache has exclusive ownership rights to the block.

Explanation:

The basic MSI (Modified, Shared, Invalid) protocol faces challenges due to the non-atomic nature of its operations, which can lead to potential issues such as deadlock and race conditions in multiprocessor systems. To address these issues, solutions like holding the bus until invalidate signals are processed help in managing coherence. Extensions like the MESI protocol introduce additional states (e.g., Exclusive and Owned) to optimize coherence management, ensuring efficient and synchronized access to shared memory blocks across caches. These enhancements improve the overall performance and reliability of cache coherence protocols in multiprocessor environments.

Snooping Cache Variations: MESI Protocol

The MESI (Modified, Exclusive, Shared, Invalid) protocol is an enhancement of the basic MSI protocol used in cache coherence, offering four distinct states to manage data across multiple caches efficiently:

- **Modified (M):** This state indicates that the cache holds a dirty copy of the block. It is exclusive to this cache, meaning no other caches have this block. The data can be both read from and written to by the processor in this cache.
- **Exclusive (E):** In the Exclusive state, the cache holds a clean copy of the block. It is exclusive to this cache, implying that no other caches currently have this block. The data can be read from by other caches but can only be written to by this cache without requiring any updates to other caches.
- **Shared (S):** This state denotes that the cache holds a clean copy of the block, which may also exist in other caches. The data is read-only in this cache to maintain coherence across all caches that share this block.
- **Invalid (I):** The Invalid state indicates that the cache block is not valid, meaning it does not currently hold any meaningful data. This state typically occurs when the cache line is initially allocated or when it has been invalidated due to updates or replacements.

The MESI protocol facilitates efficient cache coherence by allowing caches to manage the state of data blocks dynamically as they are accessed and modified across multiple processors. It minimizes unnecessary data transfers and ensures that each cache operates with consistent and up-to-date data while maintaining performance in multiprocessor systems.

In both S and E , the memory has an up-to-date version of the data

A write to a E block does not require to send the invalidation signal on the bus, since no other copies of the block are in cache.

A write to a S block implies the invalidation of the other copies of the block in cache

The Problem of Memory Consistency

- **What is consistency?**
  - When must a processor see the new value of data updated by another processor?
- **Example Scenario:**

```
P1: A = 0;          |  P2: B = 0;
  .....             |      .....
  A = 1;            |      B = 1;
  L1: if (B == 0) ; |      L2: if (A == 0) ...
```

  - Is it impossible for both `if` statements L1 & L2 to be true?

```
Yes, it is impossible for both if statements L1 & L2 to be true simultaneously in a consistent

In L1: if (B == 0), P1 is checking the value of B.
In L2: if (A == 0), P2 is checking the value of A.
According to the scenario:

P1 writes A = 1 after initially setting A = 0.
P2 writes B = 1 after initially setting B = 0.
For L1 to be true, B must be 0 at the time of the check. For L2 to be true, A must be 0 at the
```

  - What if write invalidate is delayed and processor continues?

```
If write invalidate (or any memory operation) is delayed and the processor continues execution
```

- **Memory consistency models:**
  - What are the rules for such cases?

```
Memory consistency models define the rules regarding when a processor must see the new value o
Sequential consistency: Requires that all operations from all processors appear to execute in a
```

```
Weak consistency: Allows for relaxed ordering of operations but ensures certain synchronizatio
Release consistency: Defines specific synchronization points (acquire and release operations) t
Strong consistency: Ensures that the order of operations is maintained strictly across all proc
```

Sequential Consistency

arbitrary order-preserving interleaving of memory references of sequential programs

"A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program" -Leslie Lamport

The Problem of Memory Consistency

Memory consistency refers to the order and visibility of updates to shared data among multiple processors or threads in a parallel computing system. Inconsistent memory operations can lead to incorrect program behavior, especially in concurrent programs where multiple processors or threads access and modify shared data simultaneously.

Key Points:

- **Sequential Consistency**: This is the ideal memory consistency model where all operations from all processors appear to execute in a sequential order. It ensures that the results of any execution are equivalent to some sequential order of the operations of all the processors.
- **Synchronization**: In synchronized programs, all access to shared data are ordered by synchronization operations, such as locks and releases. Here's how synchronization ensures memory consistency:

  ```
  write(x)
  ...
  release(s) {unlock}
  ...
  acquire(s) {lock}
  ...
  read(x)
  ```

  - **write(x)**: Writes data `x` to memory.
  - **release(s)**: Releases a synchronization object `s`, typically a lock, allowing other threads to acquire it.
  - **acquire(s)**: Acquires the synchronization object `s`, typically a lock, ensuring exclusive access to shared data.
  - **read(x)**: Reads data `x` from memory.

These synchronization points (release and acquire operations) ensure that memory accesses to shared data are properly ordered and consistent across multiple threads or processors. By enforcing these synchronization rules, programs can avoid race conditions and maintain correct behavior even in parallel execution environments.

Conclusion: While achieving sequential consistency across all processors can impose performance overhead in certain cases, modern synchronization mechanisms like locks, semaphores, and barriers provide effective ways to manage shared data accesses while maintaining memory consistency. Designing programs with proper synchronization mechanisms ensures that shared data accesses are properly ordered and synchronized, preventing potential issues of memory inconsistency and data races.

Synchronization

Synchronization is essential in concurrent systems, including uniprocessor systems, to manage shared resources effectively and ensure correct program behavior. There are two primary classes of synchronization:

- **Producer-Consumer**: In this synchronization pattern, a consumer process must wait until a producer process has produced data. This pattern ensures that consumers do not attempt to access data that has not been produced or is still being produced.
- **Mutual Exclusion**: This synchronization mechanism ensures that only one process can use a resource at any given time. It prevents concurrent access to shared resources that could lead to inconsistent or incorrect results.

These synchronization mechanisms are fundamental for managing shared data access and resource usage in multi-threaded and multi-processor systems, ensuring that operations are properly ordered and executed to maintain program correctness and avoid race conditions.

ISA Support for Mutual Exclusion Locks

In computer architectures, supporting mutual exclusion (mutex) locks is crucial for ensuring that only one process accesses a shared resource at a time. Here's how ISA (Instruction Set Architecture) provides support for implementing mutex locks:

- **Regular Loads and Stores**: In sequentially consistent (SC) memory models, basic load and store instructions, along with memory fences in weaker models, can be used to implement mutual exclusion. However, this approach often leads to inefficient and complex code.
- **Atomic Read-Modify-Write (RMW) Instructions**: To simplify and streamline the implementation of mutex locks, many ISAs include specific atomic RMW instructions. These instructions perform a read, modify, and write operation atomically, ensuring that the operation appears to occur instantaneously and without interruption from other processes or threads.
- **Examples of Atomic RMW Instructions**:
  - **Test and Set**: This instruction reads the current value at memory location `M[a]`, sets it to a specific value (often `1` to indicate locked state), and returns the original value.

    ```
    reg_x = M[a];
    M[a] = 1;
    ```

  - **Swap**: This instruction swaps the value at memory location `M[a]` with the value in register `reg_y`.

    ```
    reg_x = M[a];
    M[a] = reg_y;
    ```

These atomic RMW instructions ensure that operations critical for implementing mutex locks are executed atomically and efficiently within the ISA, thereby facilitating robust synchronization mechanisms in concurrent programming.