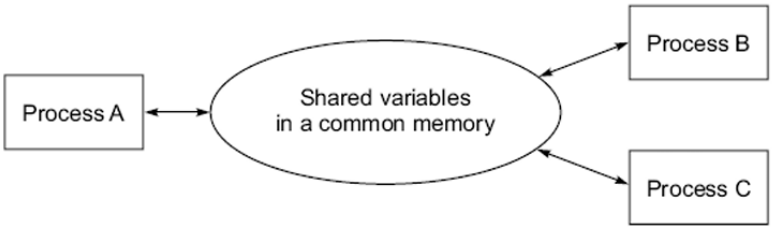
1) Define parallel programming models. Discuss any two models.

Parallel programming models provide a simplified and transparent view of the computer hardware and software system, designed specifically for multiprocessors, multicomputers, or vector/SIMD computers. These models abstract the complexity of the underlying architecture, making it easier to develop parallel programs.

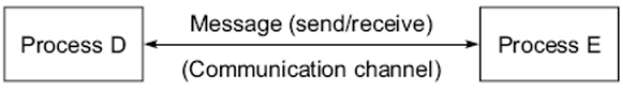
Shared-Variable Model

In the shared-variable model, multiple processes share a common address space and communicate by reading and writing shared variables. This model is commonly used in multiprocessor programming.

- **Characteristics**:
  - **Active and Passive Resources**: Processors are active resources, while memory and I/O devices are passive resources.
  - **Inter-Process Communication (IPC)**: Parallelism depends on how IPC is implemented. The process address space is shared among processes.
  - **Critical Section**: A critical section (CS) is a code segment that accesses shared variables and must be executed by only one process at a time.
    - **Mutual Exclusion**: Ensures that at most one process is executing the CS at a time.
    - **No Deadlock**: Ensures that there is no circular wait among processes.
    - **No Preemption**: Ensures that a process in CS is not interrupted until it finishes.
    - **Eventual Entry**: Ensures that each process gets a chance to enter the CS eventually.
  - **Protected Access**: The granularity of CS affects performance. Too large a CS limits parallelism, while too small a CS adds unnecessary complexity.



(a) IPC using shared variable



(b) IPC using message passing

- **Operational Modes**:
  - **Multiprogramming**: Multiple programs are executed concurrently.

- **Multiprocessing**: Multiple processors execute multiple programs simultaneously.
- **Multitasking**: Multiple tasks (processes) are executed concurrently.
- **Multithreading**: Multiple threads within a process are executed concurrently.

Message-Passing Model

The message-passing model is used primarily in multicomputer systems where each processor has its own local memory. Processes communicate by passing messages.

- **Characteristics**:
  - **Communication**: Processes residing on different processor nodes communicate by passing messages through a direct network. Messages can be instructions, data, synchronization signals, or interrupt signals.
  - **Synchronous Message Passing**:
    - No shared memory or mutual exclusion.
    - Sender and receiver processes must synchronize, similar to a telephone call.
    - No buffer is used; one process must block if the other is not ready.
  - **Asynchronous Message Passing**:
    - No synchronization in time and space is required.
    - Arbitrary communication delays may occur.
    - Similar to a postal service using mailboxes; no synchronization between senders and receivers.

Data-Parallel Model

The data-parallel model is used in SIMD computers where parallelism is achieved through hardware synchronization and flow control. This model is particularly useful for applications with regular data structures, like arrays.

- **Characteristics**:
  - **Data Parallelism**: This technique is used in array processors (SIMD).
  - **Array Language Extensions**: Various data-parallel languages are used to represent high-level data types.
  - **Example Languages**: CFD for Illiac 4, DAP Fortran for Distributed Array Processor, C* for Connection Machine.
  - **Challenge**: Matching the problem size with the machine size.

Object-Oriented Model

The object-oriented model allows objects to be dynamically created and manipulated, with processing performed by sending and receiving messages among objects. It leverages the concepts of abstraction and reusability.

- **Concurrent Object-Oriented Programming (COOP)**:
  - **Concurrent Manipulation**: Objects encapsulate data and operations and are manipulated concurrently.
  - **Actor Model**: A framework for COOP where actors (independent components) communicate via asynchronous message passing.
    - **Primitives**: Create, send-to, and become.
  - **Patterns for Parallelism**: Pipeline concurrency, divide and conquer, cooperative problem solving.

Functional and Logic Model

This model includes functional programming languages and logic programming languages, which are used for different types of parallelism.

- **Functional Programming**:
  - Examples: Lisp, Sisal, Strand 88.
  - Characteristics: No side effects, no storage, assignment, or branching; single assignment and data flow language.
- **Logic Programming**:
  - Examples: Concurrent Prolog, Parlog.
  - Characteristics: Implicit search strategy, support for parallel execution, and parallel reduction techniques; used in artificial intelligence for knowledge processing from large databases.

2) With a neat diagram, illustrate different phases of parallelizing compiler.

A parallelizing compiler transforms sequential code into parallel code, optimizing it for execution on parallel architectures. The main phases of a parallelizing compiler can be illustrated as follows:

1. **Source Code**: The starting point, consisting of the original sequential code written by the programmer.
2. **Flow Analysis**: This phase involves analyzing the data and control dependencies within the code.
   - **Data Dependence**: Determines how data dependencies between instructions affect parallelization.

- **Control Dependence**: Analyzes the order of execution of instructions.
- **Reuse Analysis**: Identifies opportunities to reuse data to improve performance.

3. **Program Optimizations**: Optimizes the code for better parallel execution.
   - **Vectorization**: Converts scalar operations to vector operations to exploit data-level parallelism.
   - **Parallelizations**: Transforms code to run concurrently on multiple processors.
   - **Locality**: Optimizes data access patterns to enhance cache performance.
   - **Pipelining**: Arranges computations to overlap execution phases, improving throughput.
4. **Parallel Code Generation**: Generates parallel code tailored to the target architecture.
   - **Granularity**: Determines the size of tasks to balance load and minimize overhead.
   - **Degree of Parallelism**: Establishes how many tasks can be executed in parallel.
   - **Code Scheduling**: Arranges the execution order of tasks to optimize performance.
5. **Target Architecture Specific Optimization**:
   - **Superscalar Processor**: Involves scheduling, register allocation, and context switching.
   - **Shared-Memory Multiprocessor**: Focuses on partitioning, synchronization, and load balancing.
   - **Distributed-Memory Multicomputer**: Handles distributed data and computations, along with message-passing for communication.

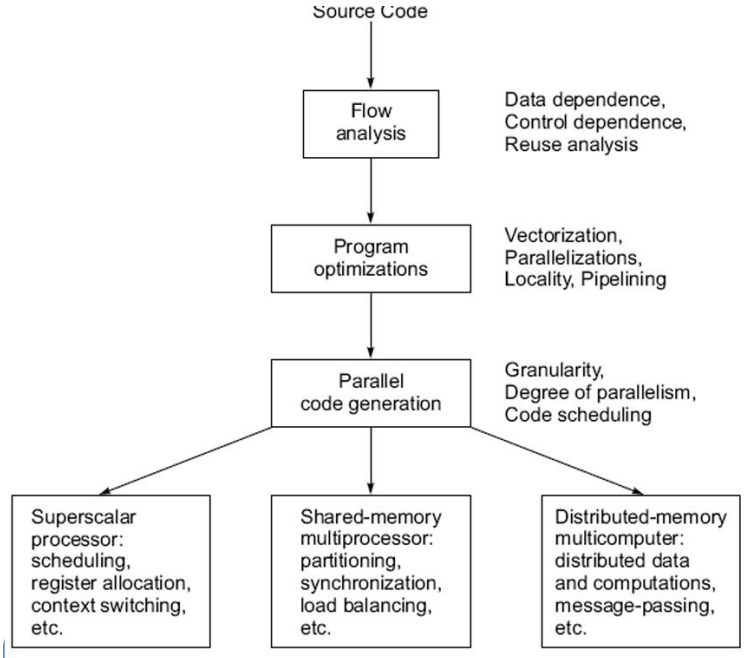


Fig. 10.4 Compilation phases in parallel code generation

3) Describe testing algorithm for dependence testing.

Dependence testing is crucial in parallel computing to identify dependencies among instructions or data elements that could potentially hinder parallel execution. The following steps outline a typical algorithm for dependence testing:

1. **Identify Dependencies:**
   - **Data Dependence (RAW, WAW, WAR):** Determine Read-After-Write (RAW), Write-After-Write (WAW), and Write-After-Read (WAR) dependencies among instructions.
   - **Control Dependence:** Check if the execution of instructions depends on control structures like branches.
   - **Resource Dependence:** Ensure the availability of resources needed for instruction execution.
2. **Flow Analysis:**
   - **Instruction Level:** Analyze the flow of instructions to identify dependencies at the VLSI or superscalar processor level.
   - **Loop Level:** Analyze dependencies within loops, especially for SIMD and systolic arrays.
   - **Task Level:** Evaluate dependencies among tasks in multiprocessor or multicomputer environments.
3. **Optimization Phases:**
   - **Local Optimization:** Optimize code within a basic block.
   - **Global Optimization:** Optimize across basic blocks considering the entire program flow.
4. **Register Renaming:**
   - **Avoiding WAW and WAR:** Use register renaming to avoid write-after-write and write-after-read dependencies. For example, if two instructions write to the same register, assign a new register to one of the instructions to prevent the WAW dependency.
5. **Operand Forwarding:**
   - **Speeding Up Execution:** Implement operand forwarding to reduce the impact of RAW dependencies. This technique allows the result of an operation to be used by subsequent instructions without writing it back to a register first.
6. **Control Dependencies Handling:**
   - **Branch Prediction:** Use accurate branch prediction techniques to minimize the performance impact of control dependencies. Mis-predicted branches should trigger a pipeline flush and reorder buffer clear.
7. **Dependence Graphs:**
   - **Graph Construction:** Construct dependence graphs to visualize and analyze dependencies among instructions. This aids in identifying and resolving conflicts.

8. **Parallel Code Generation:**
    - **Compiler Directives:** Utilize compiler directives to generate parallel code and optimize performance. Tools like Parafase and PFC can transform sequential programs into parallel ones, perform inter-procedure flow analysis, and generate vector code.

4) Illustrate the dynamic scheduling of a pipeline using Tomasulo's algorithm.

Tomasulo's algorithm is a technique used to dynamically schedule instructions to improve the utilization of the pipeline and minimize delays caused by various hazards (data hazards, structural hazards, etc.). It helps in achieving out-of-order execution while maintaining correct program execution.

Key Components

1. **Reservation Stations (RS):**
    - Temporary storage locations associated with each functional unit (like ALUs, multipliers, etc.).
    - Hold the instruction until it is ready to execute.
2. **Register Renaming:**
    - Uses a set of temporary registers to avoid conflicts between instructions that might use the same registers (eliminates WAR and WAW hazards).
3. **Common Data Bus (CDB):**
    - A bus that broadcasts the results from the functional units to the reservation stations and registers.
    - Ensures that once an operand is ready, it is available to all the instructions waiting for it.

Steps in Tomasulo's Algorithm

1. **Instruction Issue:**
    - Instructions are issued to reservation stations if there is a free station.
    - If the operands are available, they are fetched and stored in the reservation station.
    - If operands are not available, the reservation station records which functional unit will produce the operand (using tags).
2. **Execution:**
    - Once all operands for an instruction in a reservation station are available, the instruction is sent to the functional unit for execution.

- The reservation station keeps track of which operands are ready and which are not.

3. **Write-back:**
   - Once the functional unit completes the execution, it writes the result on the Common Data Bus (CDB).
   - All reservation stations and registers waiting for this result capture it from the CDB.
   - This allows the waiting instructions to proceed with execution.

Example

Consider the following instructions:

1. `I1: ADD R1, R2 -> R3`
2. `I2: MUL R3, R4 -> R5`
3. `I3: SUB R6, R3 -> R7`

Here, `I2` and `I3` depend on the result of `I1` because they need `R3`.

Execution Flow:

1. **Issue Stage:**
   - `I1` is issued to an ADD reservation station.
   - `I2` is issued to a MUL reservation station but is marked as waiting for `R3` (tagged as `I1`).
   - `I3` is issued to a SUB reservation station but is also marked as waiting for `R3` (tagged as `I1`).
2. **Operand Forwarding:**
   - When `I1` completes, it broadcasts the result of `R3` on the CDB.
   - Reservation stations holding `I2` and `I3` capture the value of `R3` from the CDB and update their status to "ready."
3. **Execution:**
   - `I2` now has all operands and can execute.
   - `I3` can execute after `I2` or in parallel if there are no resource conflicts.
4. **Write-back:**
   - Results of `I2` and `I3` are written back to their respective destination registers.

5) Explain Object Oriented Programming Model and its characteristics

Object Oriented Programming (OOP) Model is a programming paradigm that uses "objects" to design applications and computer programs. Objects are instances of classes, which can hold both data and methods. The characteristics of the OOP model include:

1. **Encapsulation**: This is the mechanism of hiding the internal state of an object and requiring all interaction to be performed through an object's methods. It helps in protecting the integrity of the data within the object.
2. **Abstraction**: Abstraction means dealing with the level of complexity by hiding the unnecessary details from the user. It allows focusing on the interactions at a higher level without knowing the internal complexities.
3. **Inheritance**: This feature allows a new class to inherit properties and behavior (methods) from an existing class. It provides a way to reuse code, create a hierarchical relationship between classes, and establish a polymorphic relationship.
4. **Polymorphism**: It refers to the ability of different objects to respond, each in its own way, to identical messages. It allows methods to do different things based on the object it is acting upon, promoting flexibility and integration.

Concurrent OOP: This involves the concurrent manipulation of objects, which are program entities encapsulating data and operations in a single unit. Concurrent OOP emphasizes abstraction and reusability.

Actor Model: It is a framework for concurrent OOP. In this model, "actors" are independent components that communicate via asynchronous message passing. The three primitives involved are:

- **Create**: To create new actors.
- **Send-to**: To send messages to actors.
- **Become**: To change the actor's behavior in response to a message.

Parallelism in COOP:

- **Pipeline Concurrency**: Divides a task into stages, each handled by a different process.
- **Divide and Conquer**: Breaks a problem into sub-problems, solves them independently, and combines the results.
- **Cooperative Problem Solving**: Multiple processes collaborate to solve a problem, sharing their intermediate results.

6) Explain Functional and logical models

Functional Programming Model

- **Characteristics**:
  - **No Side Effects**: Functions in functional programming should not produce side effects, meaning they do not alter any state or interact with outside systems (like modifying a global variable or performing I/O operations).
  - **No Storage, Assignment, or Branching**: There are no concepts of storage locations (variables that can be modified), assignment operations, or branching (conditional statements) as found in imperative programming languages.
  - **Single Assignment and Data Flow**: Variables are assigned exactly once. This aligns with the principles of data flow programming where the output of one operation is directly used as the input to another, promoting immutability and predictability.
- **Examples**: Languages like Lisp, Sisal, and Strand 88 are mentioned as examples of functional programming languages.

Logic Programming Model

- **Characteristics**:
  - **Knowledge Processing**: Logic programming is primarily used for knowledge representation and processing, often in the context of large databases or artificial intelligence.
  - **Implicit Search Strategy**: It supports implicit search strategies where the logic interpreter automatically searches through possible solutions to find one that satisfies the given constraints or rules.
  - **Parallel Execution**: It supports parallel execution through techniques like AND-parallelism (parallel execution of conjunctive goals) and OR-parallelism (parallel execution of alternative solutions).

7) Describe in brief Tomasula algorithm

Tomasulo's algorithm is a hardware algorithm used for dynamic scheduling of instructions to overcome some of the limitations posed by static scheduling. Here is a brief description of the algorithm based on the provided document:

1. **Register Renaming**:
   - Register renaming is an implicit part of Tomasulo's algorithm.
   - It requires a set of program-invisible registers to which programmable registers are re-mapped.
   - These invisible registers are provided with reservation stations of functional units.
2. **Functional Units and Reservation Stations**:

- Functional units are assumed to be internally pipelined and can complete one operation per clock cycle.
- Each functional unit can initiate one operation per clock cycle if the reservation station is ready with the required input operands.
- Reservation stations hold the operation code and operand values (if available) or operand tags (if not available).

3. **Operation Execution**:
   - When the operands become available, the functional unit initiates the required operation.
   - Operand values are forwarded over the common data bus (CDB) along with source tags.
   - These values are copied into all reservation station operand slots with matching tags.
4. **RAW Dependence Handling**:
   - Assume instruction I1 writes its result into R4, and subsequent instructions I2 and I3 read the result.
   - If the result from I1 is not available when I2 and I3 are issued, they are parked in the reservation stations with source tags corresponding to I1's output.
   - When I1's result is available, it is sent over the CDB with the source tag, and all destinations with matching tags copy the data simultaneously.
5. **Dynamic Scheduling**:
   - Instructions are scheduled dynamically, considering inter-instruction dependences and the state of functional units.
   - This maximizes instruction-level parallelism.

Tomasulo's algorithm effectively manages data hazards and enhances instruction-level parallelism by dynamically scheduling instructions and using register renaming to resolve WAR and WAW dependences

8) Explain synchronous message passing and asynchronous passing related to message passing model

Synchronous Message Passing

1. Definition: Synchronous message passing requires that both the sender and receiver processes are synchronized in their communication. This means both processes must be ready for the communication to occur.

2. No Shared Memory: Unlike shared-memory models, synchronous message passing does not rely on shared memory. Instead, all data transfer occurs through message exchanges.

3. No Mutual Exclusion: There is no need for mechanisms to ensure mutual exclusion since there is no shared data that might lead to concurrent access issues.

4. Synchronization: Synchronization is critical in synchronous message passing. The sender process must wait until the receiver process is ready to accept the message, and vice versa. This ensures that messages are only passed when both processes are prepared for communication.

5. Blocking Behavior: A notable characteristic of synchronous message passing is its blocking behavior. If the sender process is ready to send a message but the receiver process is not ready to receive it, the sender process will be blocked (i.e., it will wait) until the receiver is ready. Similarly, the receiver process will wait if it is ready to receive a message but no message has been sent yet.

6. Example: A practical example of synchronous message passing is a telephone call where both parties must be available to communicate in real time. If one party calls and the other is not available, the caller must wait until the other party is ready to talk.

Asynchronous Message Passing

1. Definition: Asynchronous message passing allows the sender and receiver processes to operate independently without needing to synchronize their communication times. Messages can be sent at any time, and the receiver can process them later.

2. No Synchronization Required: There is no need for the sender and receiver to be synchronized in time and space. The sender can send a message without knowing if the receiver is ready, and the receiver can process the message at a later time.

3. Communication Delay: Asynchronous message passing may involve arbitrary communication delays. The sender does not necessarily know when the receiver will process the message, leading to potential delays in acknowledgment.

4. Buffering: Messages are typically stored in a buffer (or mailbox) until the receiver is ready to process them. This allows the sender to continue its execution without waiting for the receiver, enhancing concurrency and reducing blocking.

5. Non-blocking Behavior: In asynchronous message passing, the sender process does not wait for the receiver to be ready. Once the message is sent, the sender can continue its execution. The receiver retrieves the message from the buffer when it is ready to process it.

6. Example: A common example of asynchronous message passing is email communication. The sender can send an email at any time, and the receiver can read and respond to it later. There is no need for both parties to be available simultaneously.

9) Explain With neat diagram Recoder buffer

Structure of a Reorder Buffer

A reorder buffer is organized as an array of entries, each containing the following fields:

1. Instruction Identifier: Identifies the instruction.
2. Value Computed: The result of the instruction.
3. Program-Specified Destination: Where the computed value should be stored.
4. Completion Flag: Indicates whether the instruction has completed execution and its result is available (Module-5 ACA).

Role of the Reorder Buffer

- Maintaining Program Order: While instructions may complete out of order, the reorder buffer ensures they are committed in the correct program order. This means the program and processor states reflect only the results of instructions that have been committed.
- Handling Data Dependences: The reorder buffer helps in resolving data dependences. For example, Read After Write (RAW) dependences can hold up the execution of instructions waiting for their operands. The reorder buffer ensures these dependencies are managed and resolved efficiently.
- Managing Control Dependences: If a branch prediction is incorrect, instructions in the reorder buffer that are part of the mis-predicted branch are flushed to maintain correct program flow.
- Resource Dependences: The buffer allows instructions to wait for the required functional unit to become available, thus enabling out-of-order execution while ensuring resource availability

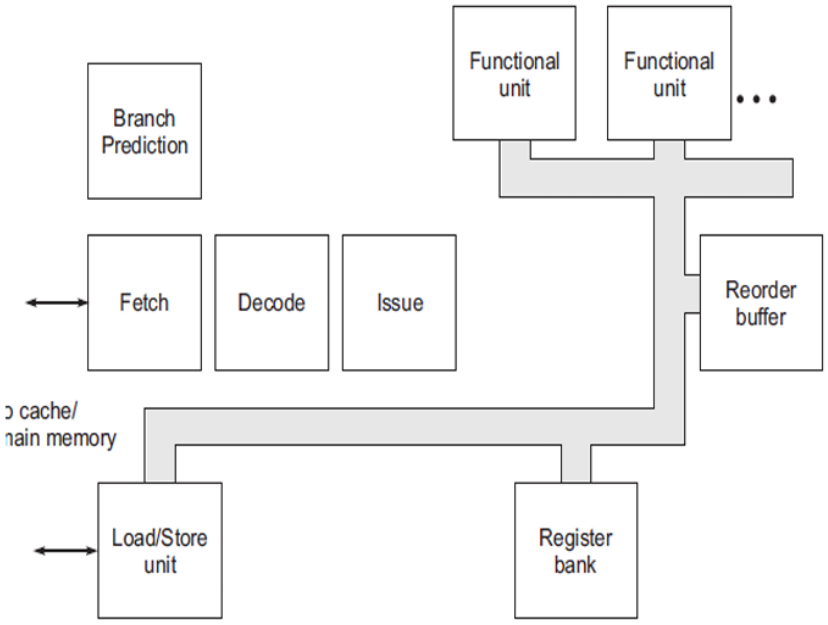


Fig. 12.4 Processor design with reorder buffer