CS533: Speculative Parallelization (Thread-Level Speculation)

Josep Torrellas

University of Illinois in Urbana-Champaign

March 5, 2015

Concepts

- Main idea: execute unsafe threads in parallel
- Typically threads come from a sequential application
- Hardware monitors for data dependence violations
- If violation: offending thread is squashed and restarted
- If no violation:
  - Thread completes
  - When all predecessor threads are completed, thread becomes non-speculative or safe
  - When all predecessor threads have committed, thread commit
- Note: thread commit one by one and in order

Communication between Threads

- Threads in a CMP communicate with each other with:
  - Registers
  - Memory
- Register communication:
  - Needs hardware between processors
  - Dependences between threads known by compiler
  - Can be producer initiated or consumer initiated
  - If consumer first:
    - consumer stalls
    - producer forwards
  - If producer first:
    - producer writes and continues
    - consumer reads later

Communication (II)

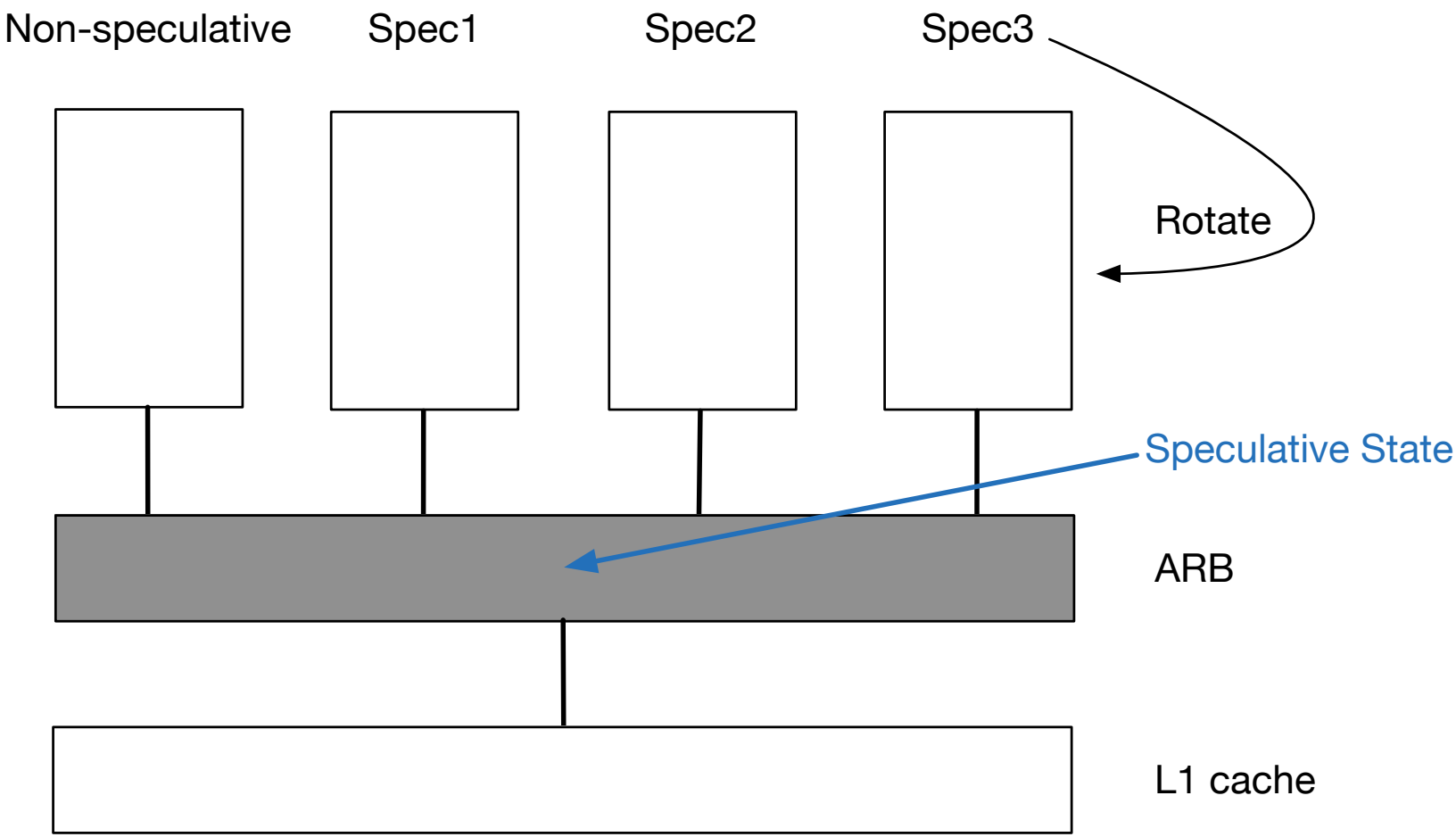
Memory communication:

- May cause thread squashing
- Not known by the compiler
- Loads are done speculatively:
  - get the data from the closest predecessor
  - keep record that read the data (L1 cache or other structure)
- Stores are done speculatively
  - buffer the update while speculative (write buffer or L1)
  - check successors for premature reads
  - if successors prematurely read: squash
  - typically squash the offending thread and all successors
- When thread commits: its state is merged with main memory (safe state)
  - write back
  - ask for owernship

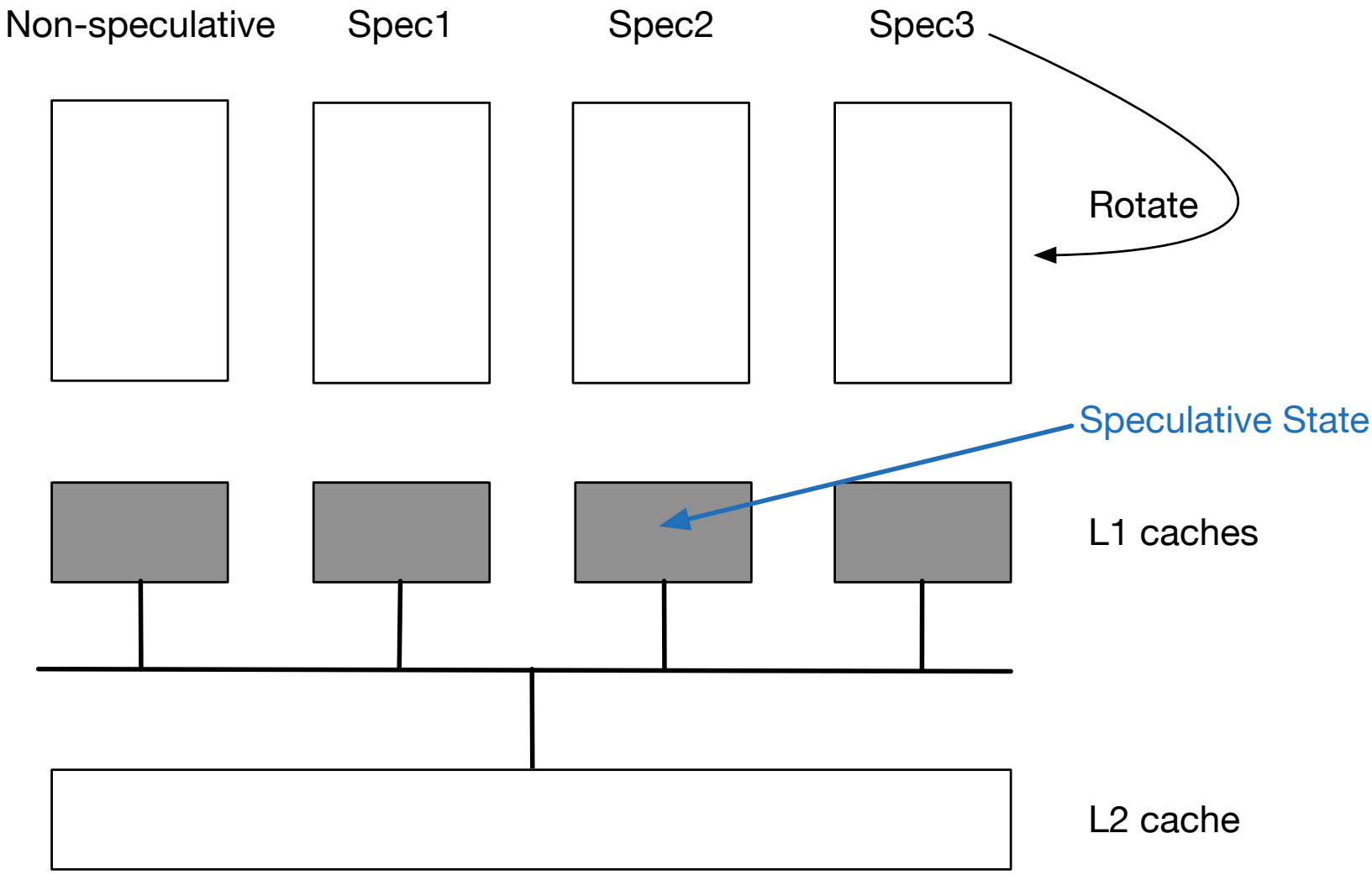
Dependence Violations

- Types of dependence violations: out of order ...
  - LD, ST: name dependence; hardware may handle
  - ST, ST: name dependence; hardware may handle
  - ST, LD: true dependence; causes a squash
- To handle name dependences: multiple-version support:
  - every spec or non-spec store to a location: creates a new version
- Keep and check for speculative versions:
  - Centralized scheme: ARB (paper by Sohi et al.)
  - De-centralized schemes: L1 or write buffers (paper by Krishnan et al.)

Centralized



Decentralized

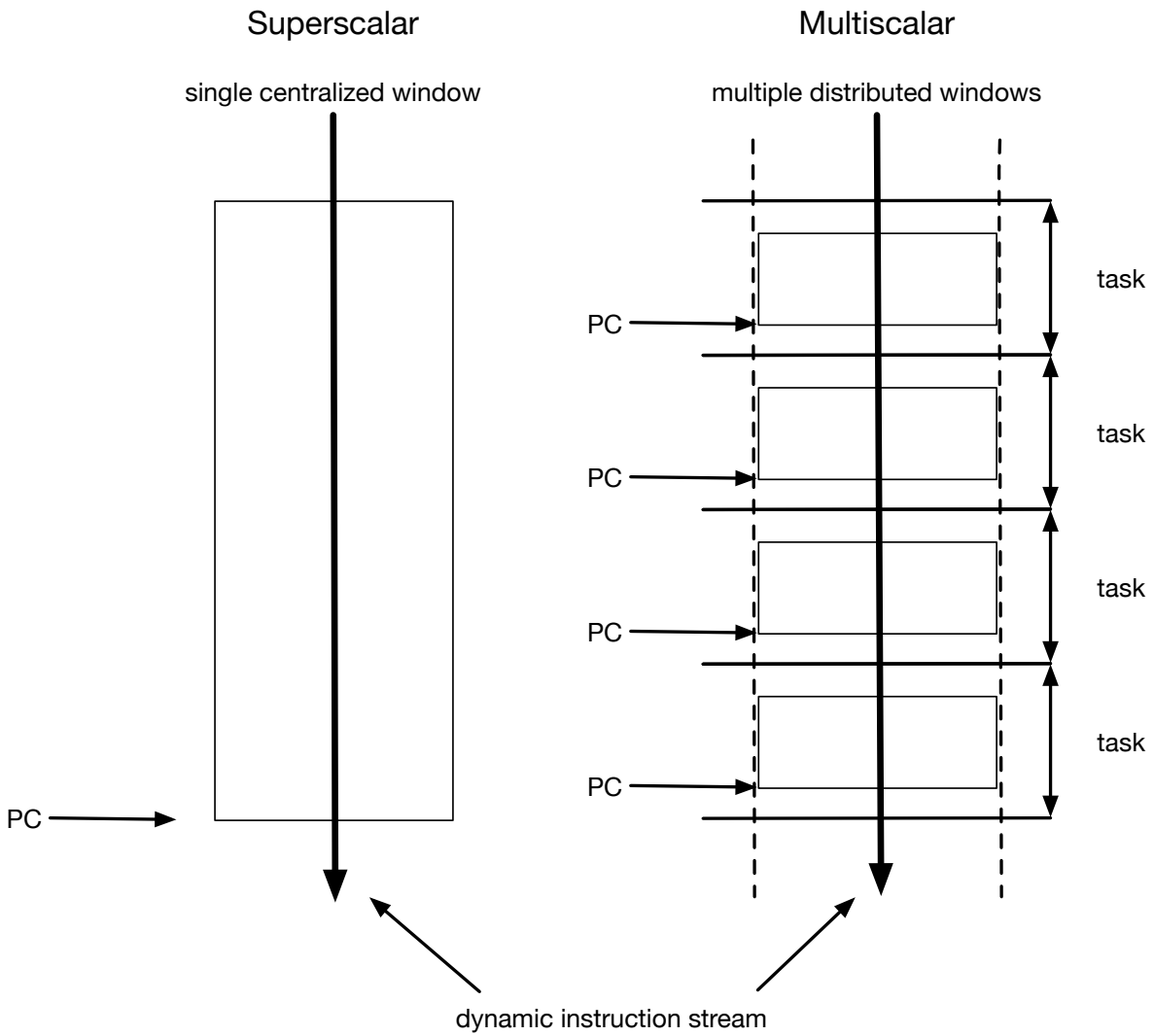


Multiscalar Overview

G. Sohi et al. "Multiscalar Processors" *ISCA*, 1995

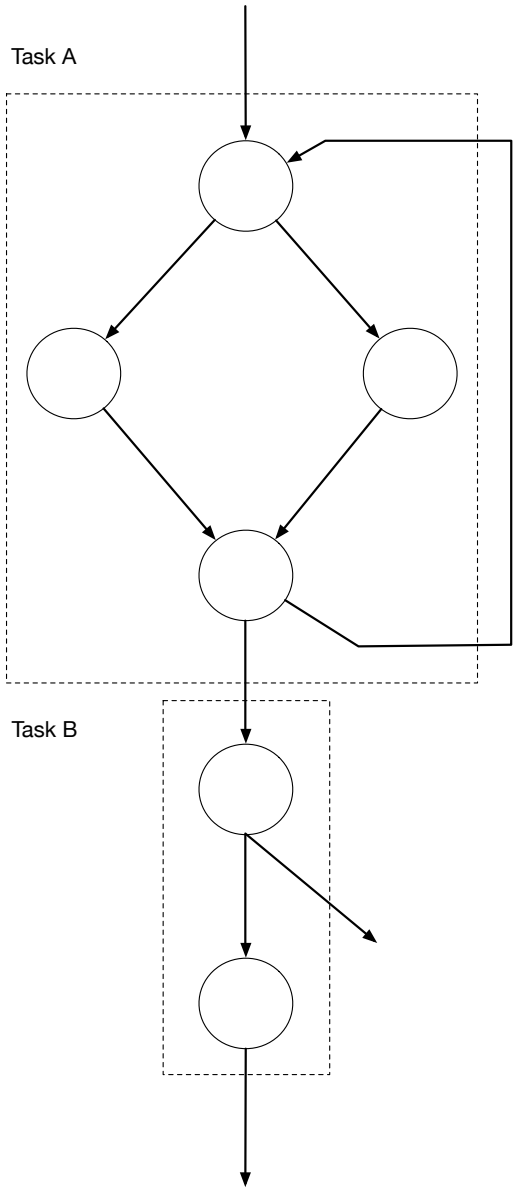
- Exploit "implict" thread-level parallelism within a serial program
- Compiler divides program into tasks
- Tasks scheduled on independent processing resources
- Hardware handles register dependences between tasks
- Memory speculation for memory dependences

Expandable Split-Window Paradigm

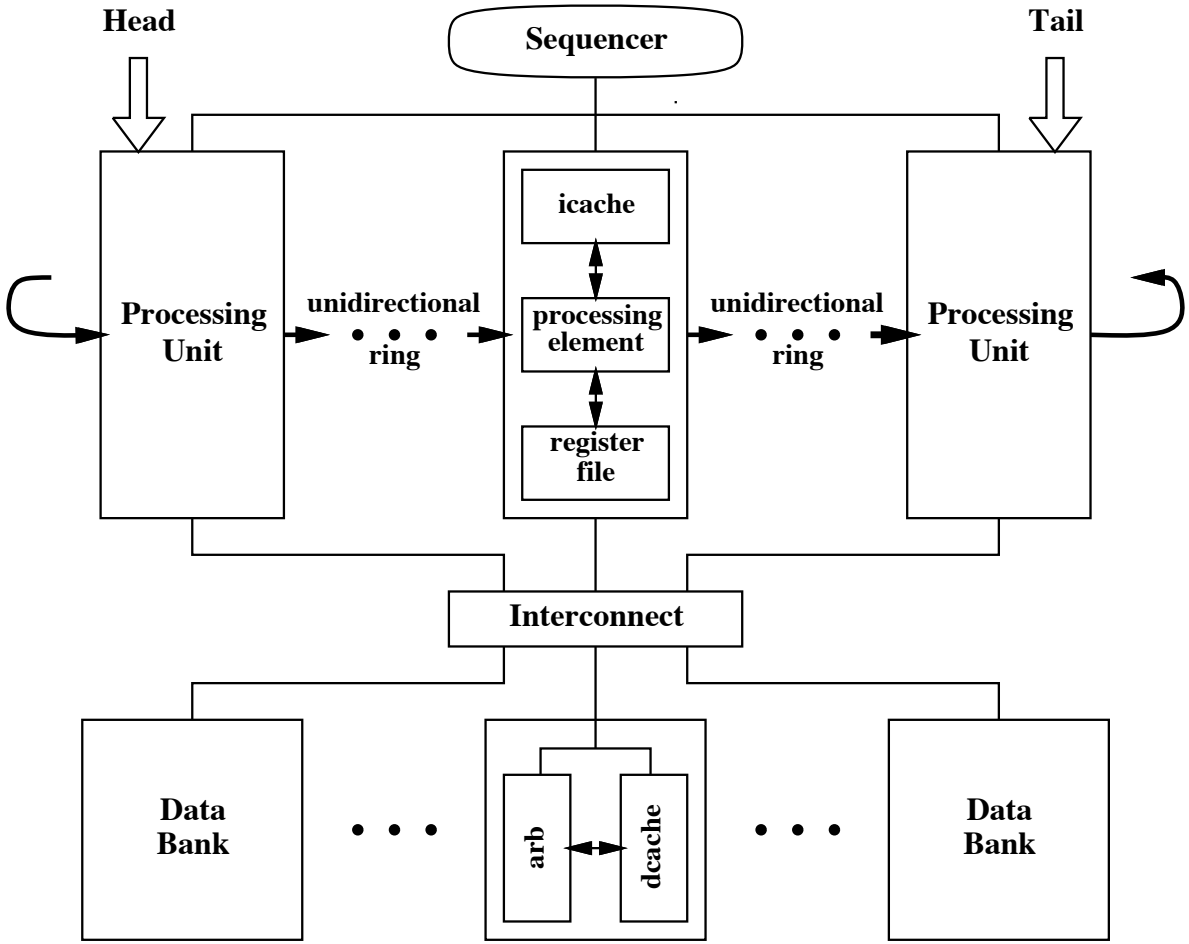


Tasks

- A task is a subgraph of the control flow graph (CFG)
  - e.g., a basic block, basic blocks, loop body, function, ...
- Tasks are selected by compiler and conveyed to hardware
- Tasks are predicted and scheduled by processor
- Tasks may have data and/or control dependences



Multiscalar Processor



Compiler's Job

Task selection – partition CFG into tasks

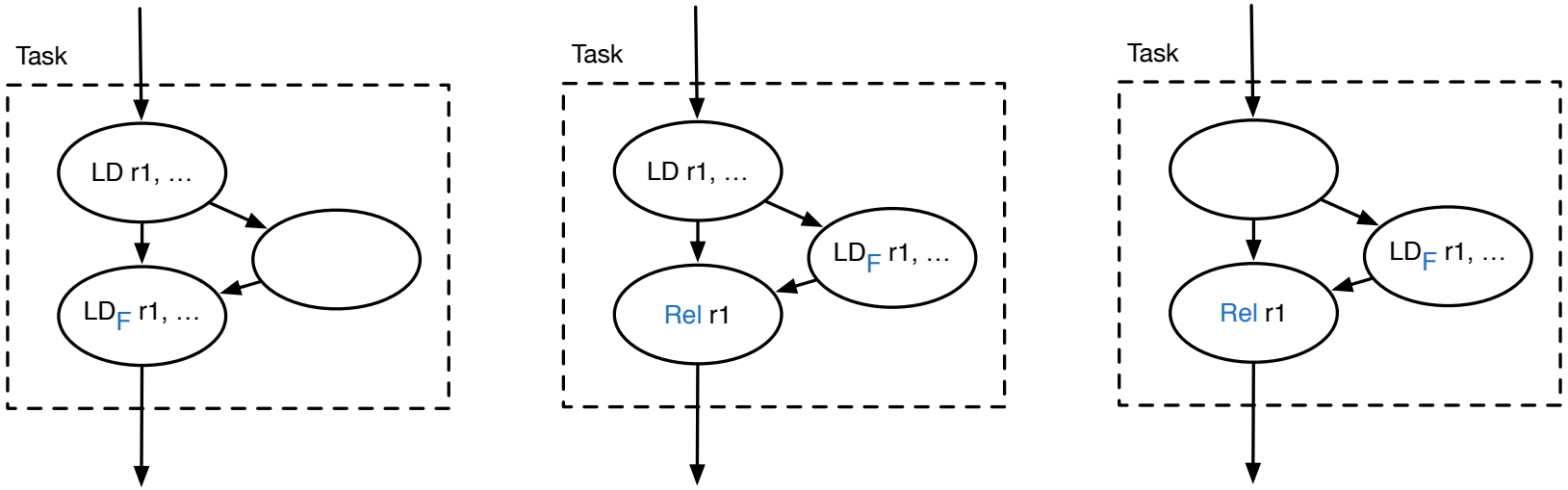
- Load balance
- Minimize inter-task data dependences
- Minimize inter-task control dependences
  - By embedding hard-to-predict branches within tasks

Convey information in the executable

- Task headers
  - `create_mask` (1 bit per register)
    - Indicates all registers that are *possibly* modified or created by the task
    - Don't forward instances received from prior tasks
  - PCs of successor tasks
- Insert special instructions and opcodes

Special Opcode Bits and Instructions

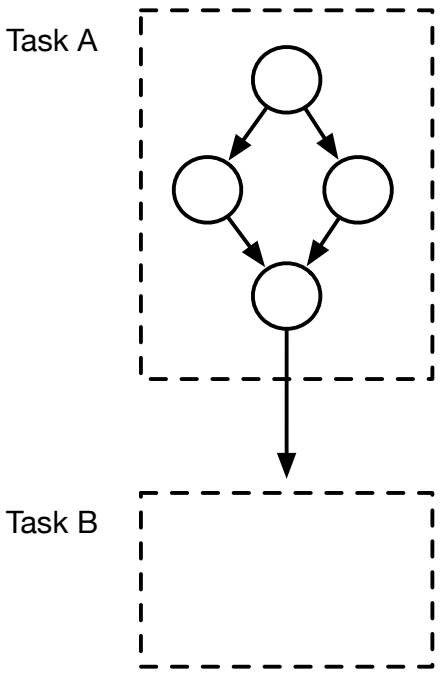
- Compiler must identify the last instance of a register within a task
  - Opcodes that write a register have additional *forward bit*, indicating the instance should be forwarded
  - Stop bits – indicate end of task
  - Release instruction
    - Release instruction tells PE to forward the value
- Examples



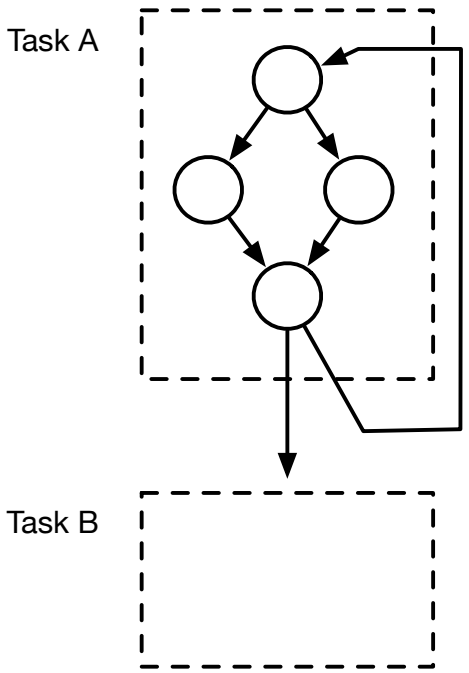
F: forward bit
Rel: release

Task Sequencer

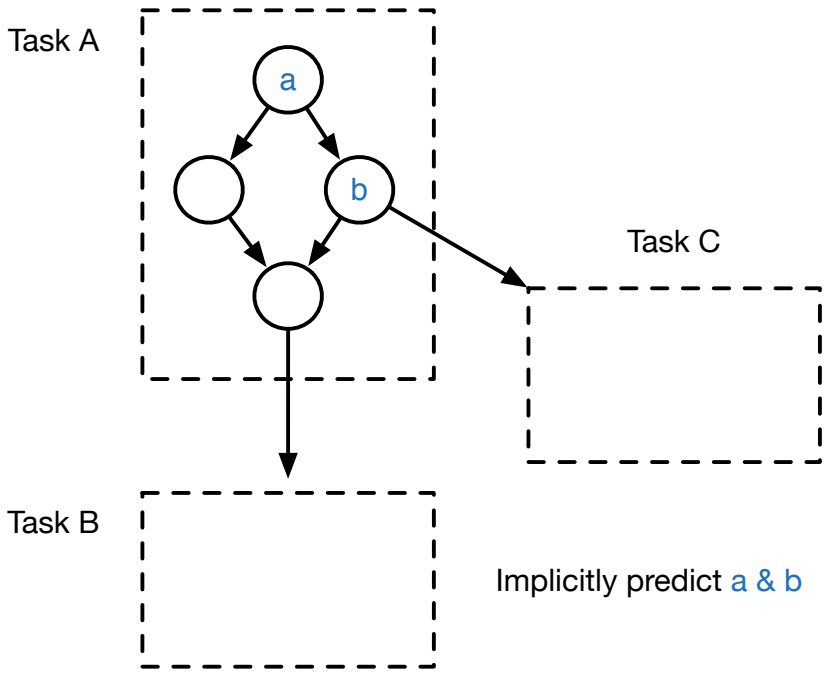
- Task prediction analogous to branch prediction
- Predict inter-task control flow



Control independent



Highly predictable
inter-task branch



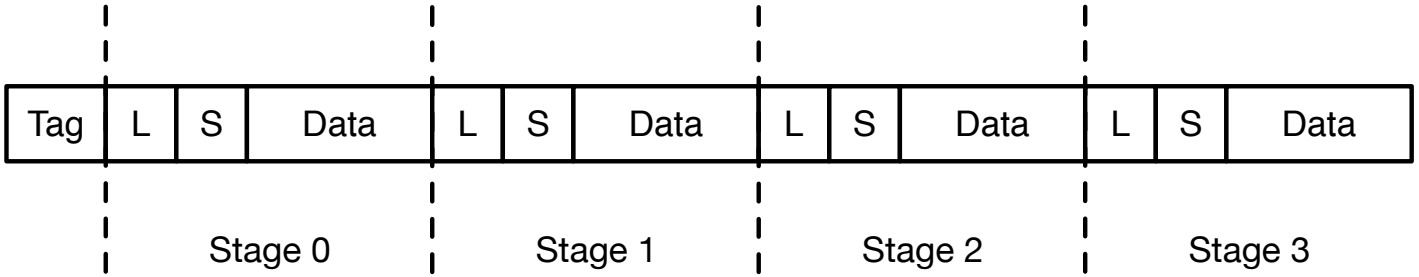
Implicitly predict a & b

Inter-task Dependences

- Control dependences
  - Predict
  - Squash subsequent tasks on misprediction
- Data dependences
  - Register file: mask bits and forwarding (stall until available)
  - Memory: address resolution buffer (speculative load, squash on violation)

Address Resolution Buffer (ARB)

- Multiscalar issues loads to ARB/D-cache as soon as address is computed
- Optimistic speculation: No prior unresolved stores to the same address
- ARB is organized like a cache, maintaining state for all outstanding load/store addresses
- An ARB entry



Stage = Task = PE
L: load performed
S: store performed
Data: store data

ARB

- Loads
  - ARB miss – data comes from D-cache (no prior stores yet)
  - ARB hit – get most recent data to the load, which may be from D-cache, or nearest prior stage with $S = 1$
- Stores
  - ARB buffers speculative stores
  - When head pointer moves to PE_i, commit all stores in stage i to the D-cache

$\rightarrow$ ARB performs memory renaming

Summary: Multiscalar

- Software and hardware extract parallelism
  - Divide CFG into tasks
  - Speculatively execute tasks
- Inter-task control dependence
  - Prediction
- Inter-task data dependence
  - (Register) `create_mask`'s and forwarding
  - (Memory) speculative load, disambiguation by ARB

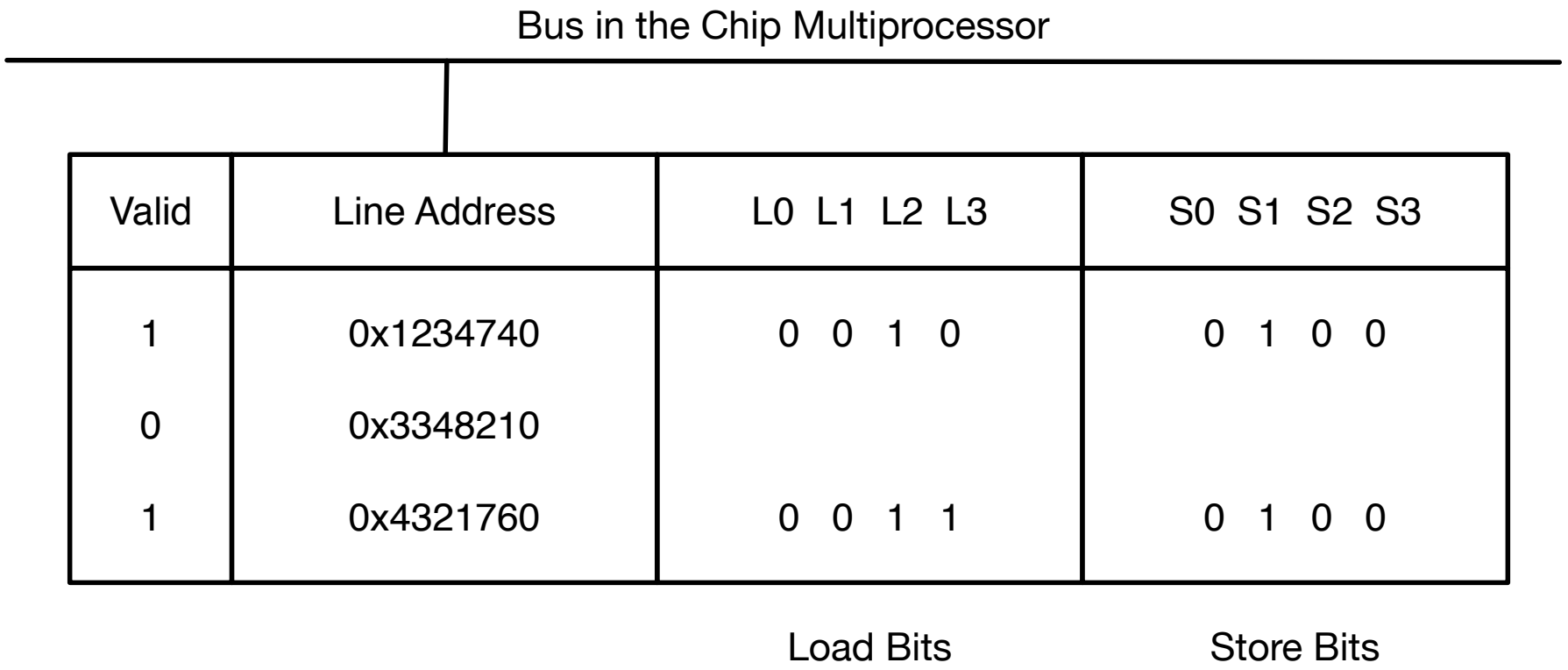
A Chip Multiprocessor Architecture with Speculative Multithreading

V. Krishna and J. Torrellas

Speculative Versioning

- A load must eventually read the value created by its most recent store:
  - A load must be squashed and re-executed if done before the store
  - All stores that folow must be buffered until the load is done
- A memory location must eventually have the correct version $\rightarrow$ speculative versions must be committed in program order
- Information on the data read/written speculatively can be kept:
  - tags of caches
  - in directory-like structure: MDT

Memory Disambiguation Table (MDT)



- When a thread becomes non-speculative $\rightarrow$ clear its bits
- Multibanked