SFU

CMPT 450/750: Computer Architecture
Fall 2021

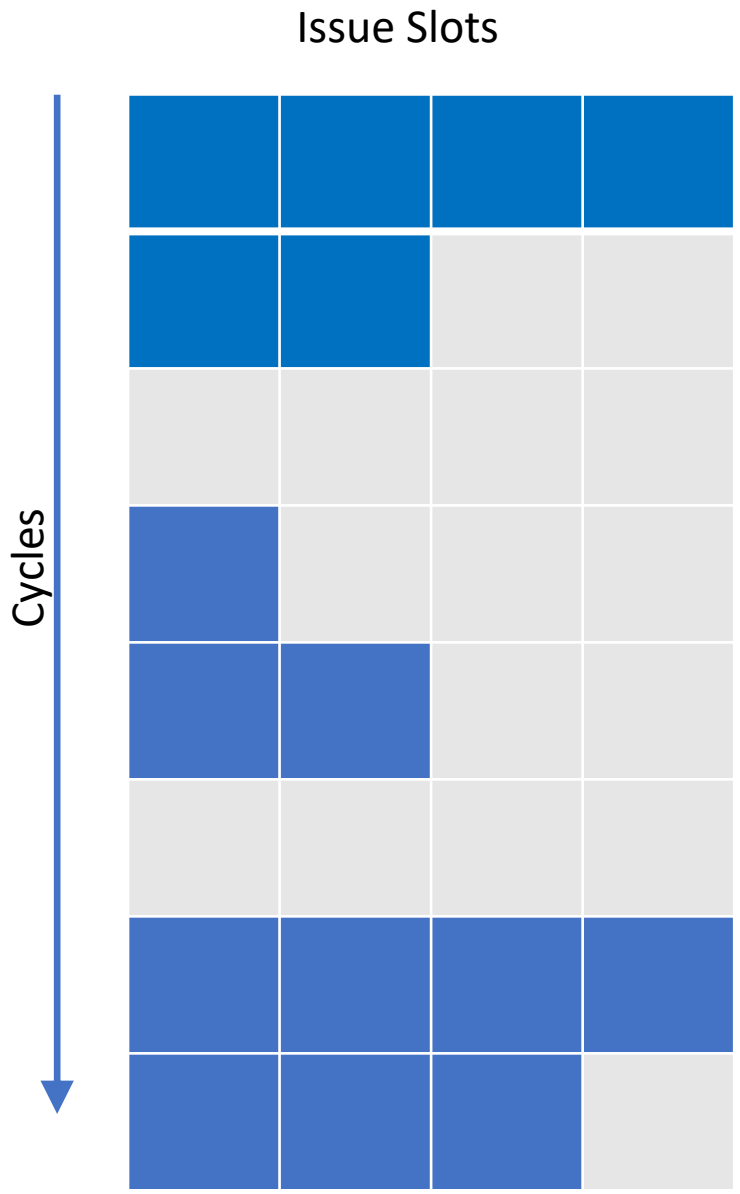
Multithreading

Alaa Alameldeen & Arrvindh Shriraman

Simultaneous Multithreading

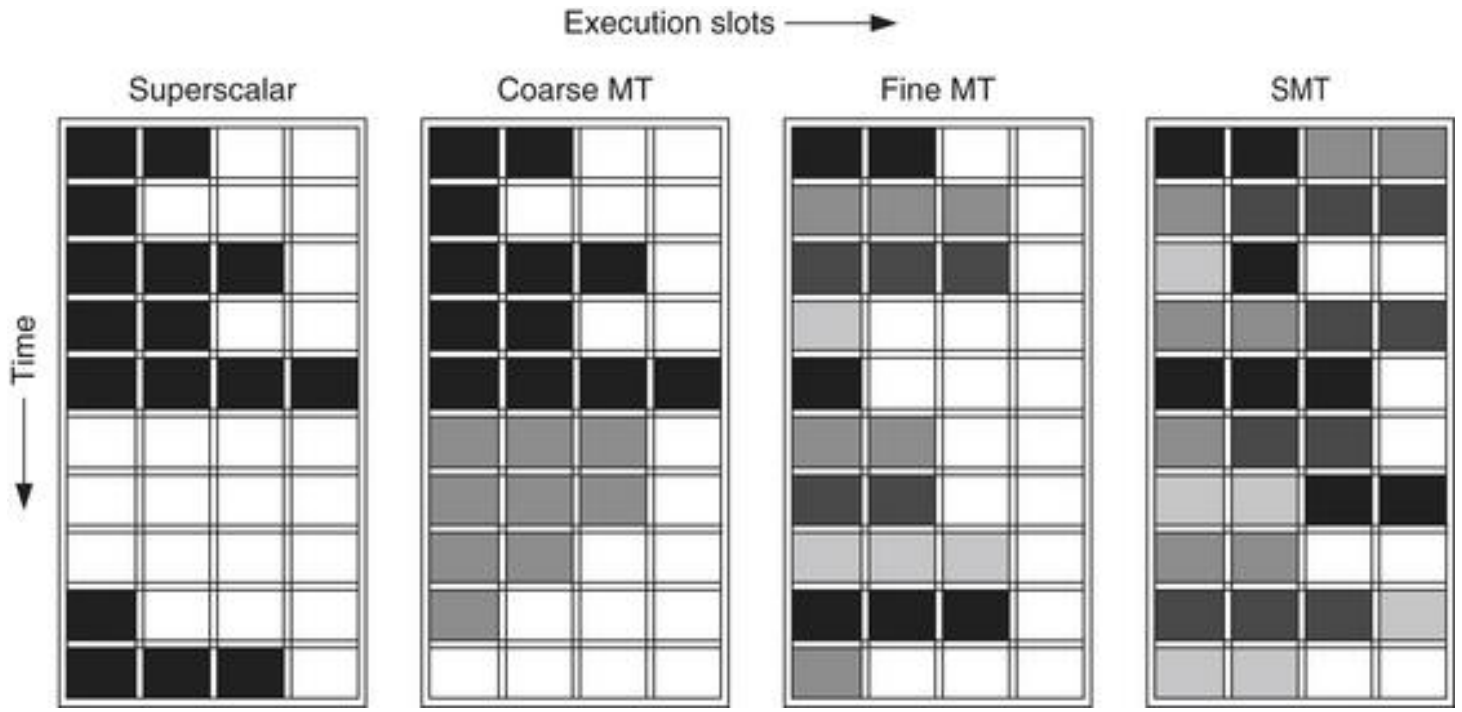
Why Multithreading?

- **ILP limitations of superscalar processors**
  - ➢Many control, data and functional dependences
- **Wide superscalar pipelines cannot use all issue slots**
  - ➢Vertical Waste: All issue slots in a cycle are not used
  - ➢Horizontal waste: Some issue slots in a cycle are not used
- **To increase throughput, we need to use thread-level parallelism (TLP)**



Different Types of Multithreading

- **Coarse-grained multithreading**
  - Switch between threads every few cycles
- **Fine-grained multithreading**
  - Switch between threads every cycle
  - Removes vertical waste
  - Still limited by ILP available within each thread
- **Simultaneous multithreading**
  - Issue instructions from multiple threads simultaneously (in the same cycle)
  - Addresses both horizontal and vertical waste



ARCH Figure 3.28

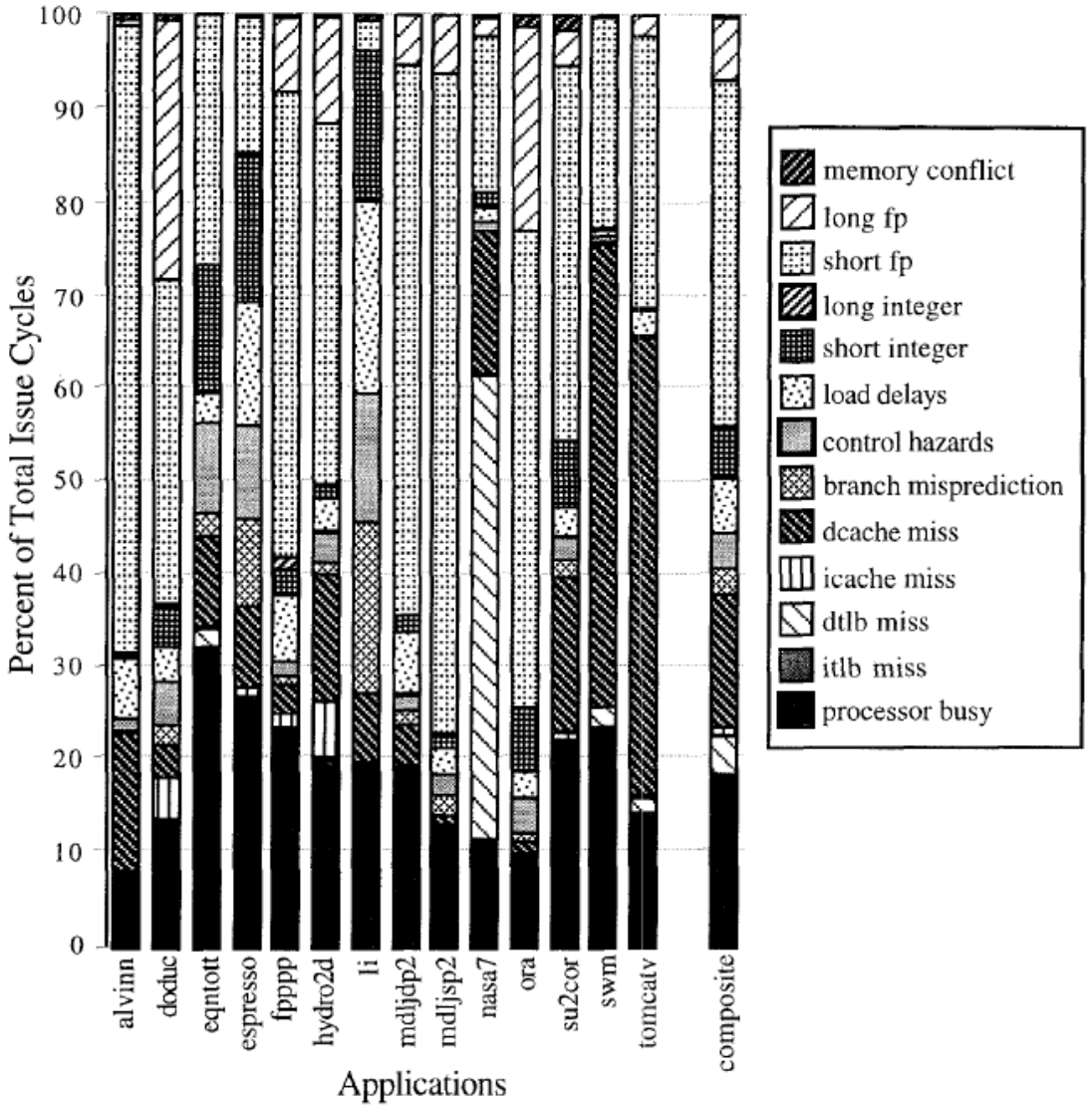
Multithreaded Programs

- **Thread vs. process**
  - Threads in a process share virtual address space
  - Processes have different virtual address spaces
- **Design Issues:**
  - Each thread needs its own set of registers (register address space is not shared)
  - Threads cause interference in instruction and data caches
  - Programs need to be parallelizable into multiple threads
  - Synchronization is necessary, may cause some threads to be idle (OS idle loop)

Superscalar Processors: Where Have Cycles Gone?

- Issue slots are utilized only 19% of the time
- Many causes for issue stall cycles (Figure)
- Need aggressive latency-hiding techniques

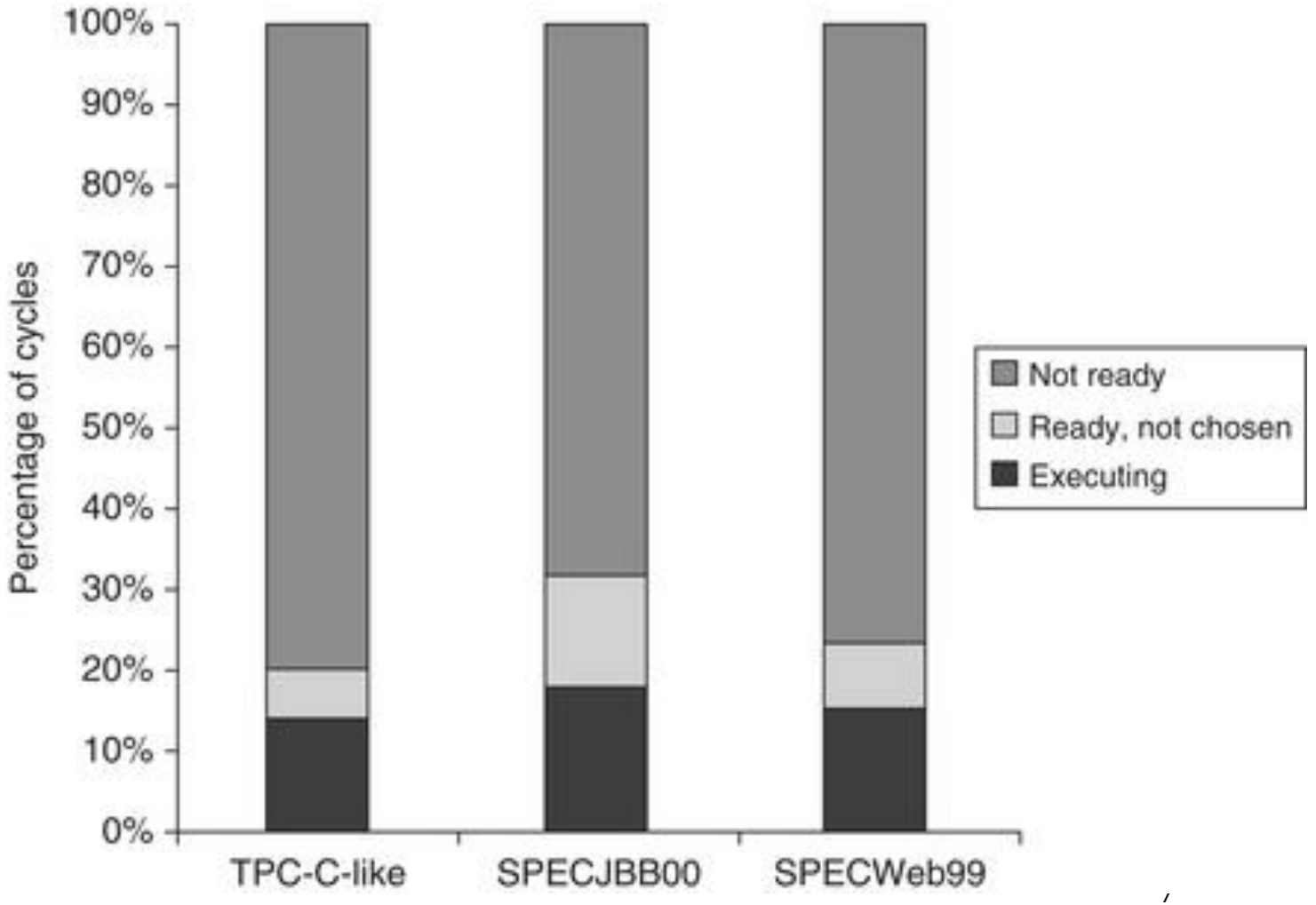
Tullsen et al., Figure 2



Commercial Multithreaded Workloads: Lost Cycles

- Threads issue instructions in a small percentage of cycles "executing"
- Threads could issue cycles but other threads use all issue resources "Ready, not chosen"
- Most of the time, threads are "Not Ready"

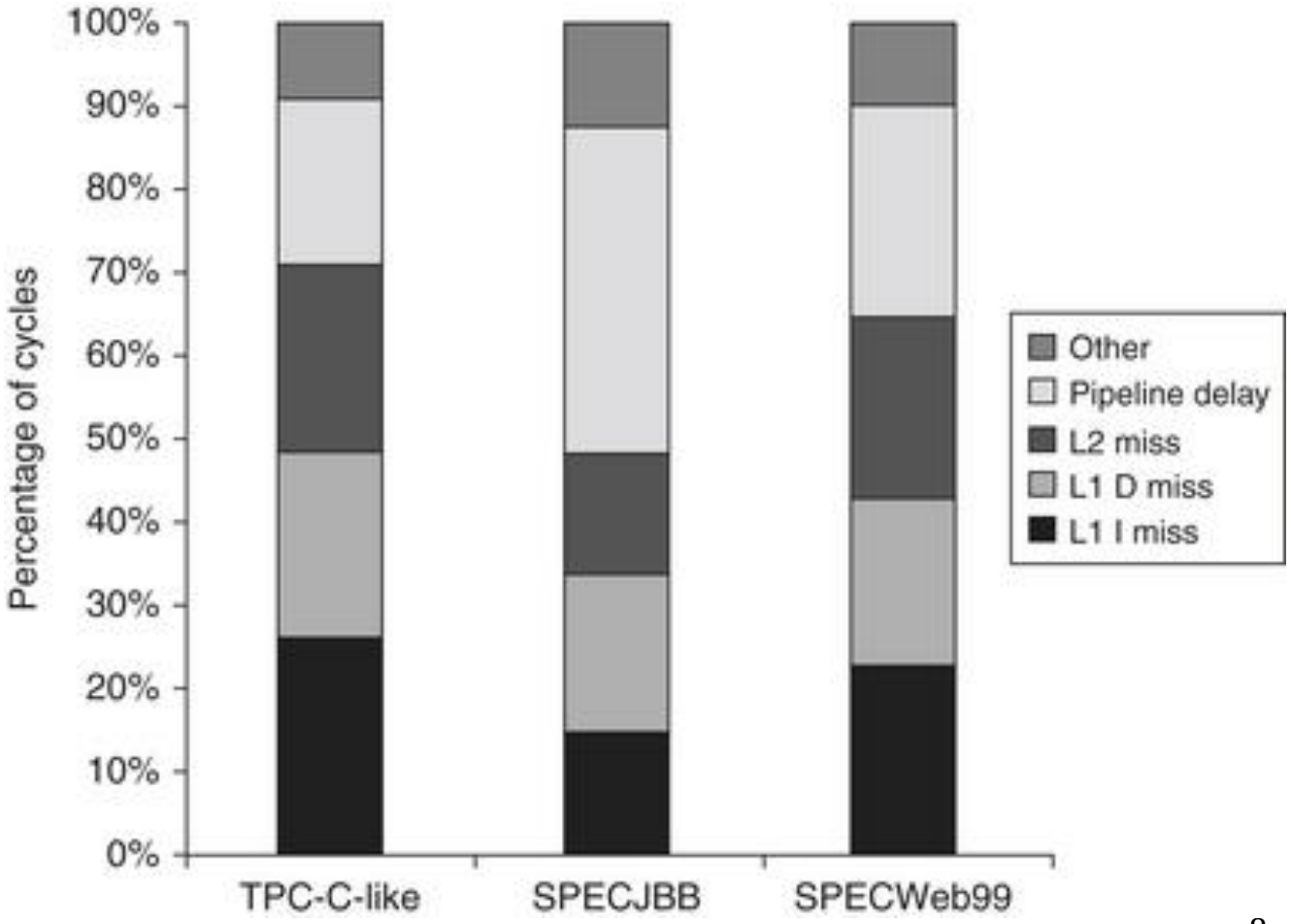
ARCH Figure 3.31



Commercial Workloads: Where Have Cycles Gone?

- Figure shows percentage of cycles lost for different reasons
- "Other" is mainly "store buffer full" in TPC-C, "atomic instructions" in jbb, both in SPECweb

ARCH Figure 3.32



Latency Hiding Techniques

Tullsen et al., Table 3

Source of Wasted Issue Slots	Possible Latency-Hiding or Latency-Reducing Technique
instruction tlb miss, data tlb miss	decrease the TLB miss rates (e.g., increase the TLB sizes); hardware instruction prefetching; hardware or software data prefetching; faster servicing of TLB misses
I cache miss	larger, more associative, or faster instruction cache hierarchy; hardware instruction prefetching
D cache miss	larger, more associative, or faster data cache hierarchy; hardware or software prefetching; improved instruction scheduling; more sophisticated dynamic execution
branch misprediction	improved branch prediction scheme; lower branch misprediction penalty
control hazard	speculative execution; more aggressive if-conversion
load delays (first-level cache hits)	shorter load latency; improved instruction scheduling; dynamic scheduling
short integer delay	improved instruction scheduling
long integer, short fp, long fp delays	(multiply is the only long integer operation, divide is the only long floating point operation) shorter latencies; improved instruction scheduling
memory conflict	(accesses to the same memory location in a single cycle) improved instruction scheduling

Simultaneous Multithreading Models

- **SM: Full Simultaneous Issue**
  - Completely flexible model: All threads compete for each of the issue slots every cycle
  - Disadvantage: Hardware complexity
- **SM: Single Issue**
  - Each thread can issue at most one instruction every cycle
- **SM: Dual Issue and SM: Four Issue**
  - Each thread can issue at most two (Dual Issue) or four (Four issue) instructions every cycle
- **SM: Limited Connection**
  - Each thread is connected to exactly one of each type of functional unit
  - Limits scheduling choices for functional units to reduce hardware complexity

Hardware Complexity of Multithreading Models

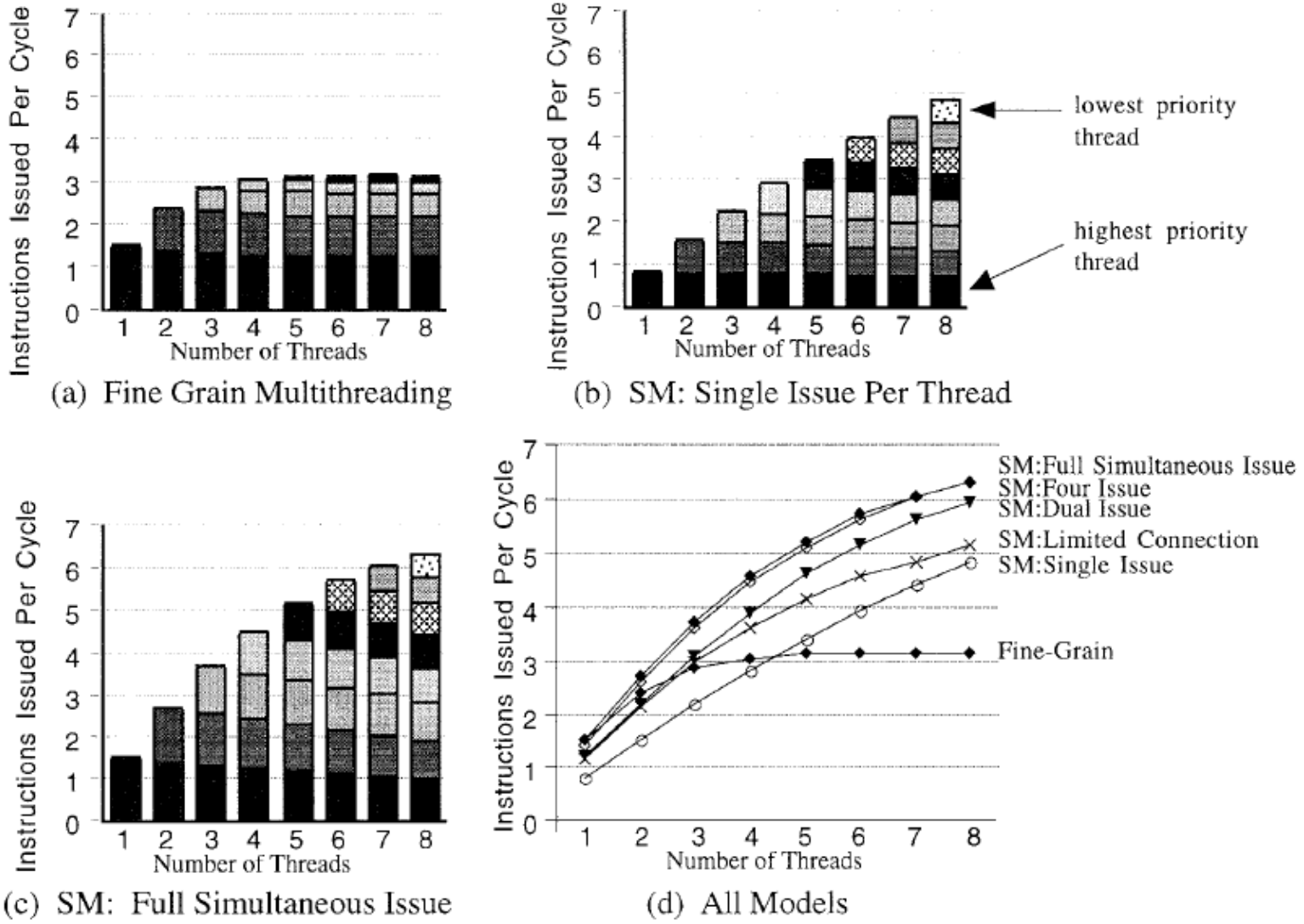
Tullsen et al., Table 4

Model	Register Ports	Inter-inst Dependence Checking	Forwarding Logic	Instruction Scheduling onto FUs	Notes
Fine-Grain	H	H	H/L*	L	Scheduling independent of other threads.
SM:Single Issue	L	None	H	H	
SM:Dual Issue	M	L	H	H	
SM:Four Issue	M	M	H	H	
SM:Limited Connection	M	M	M	M	No forwarding between FUs of same type; scheduling is independent of other FUs
SM:Full Simultaneous Issue	H	H	H	H	Most complex, highest performance

* We have modeled this scheme with all forwarding intact, but forwarding could be eliminated, requiring more threads for maximum performance

SMT Performance

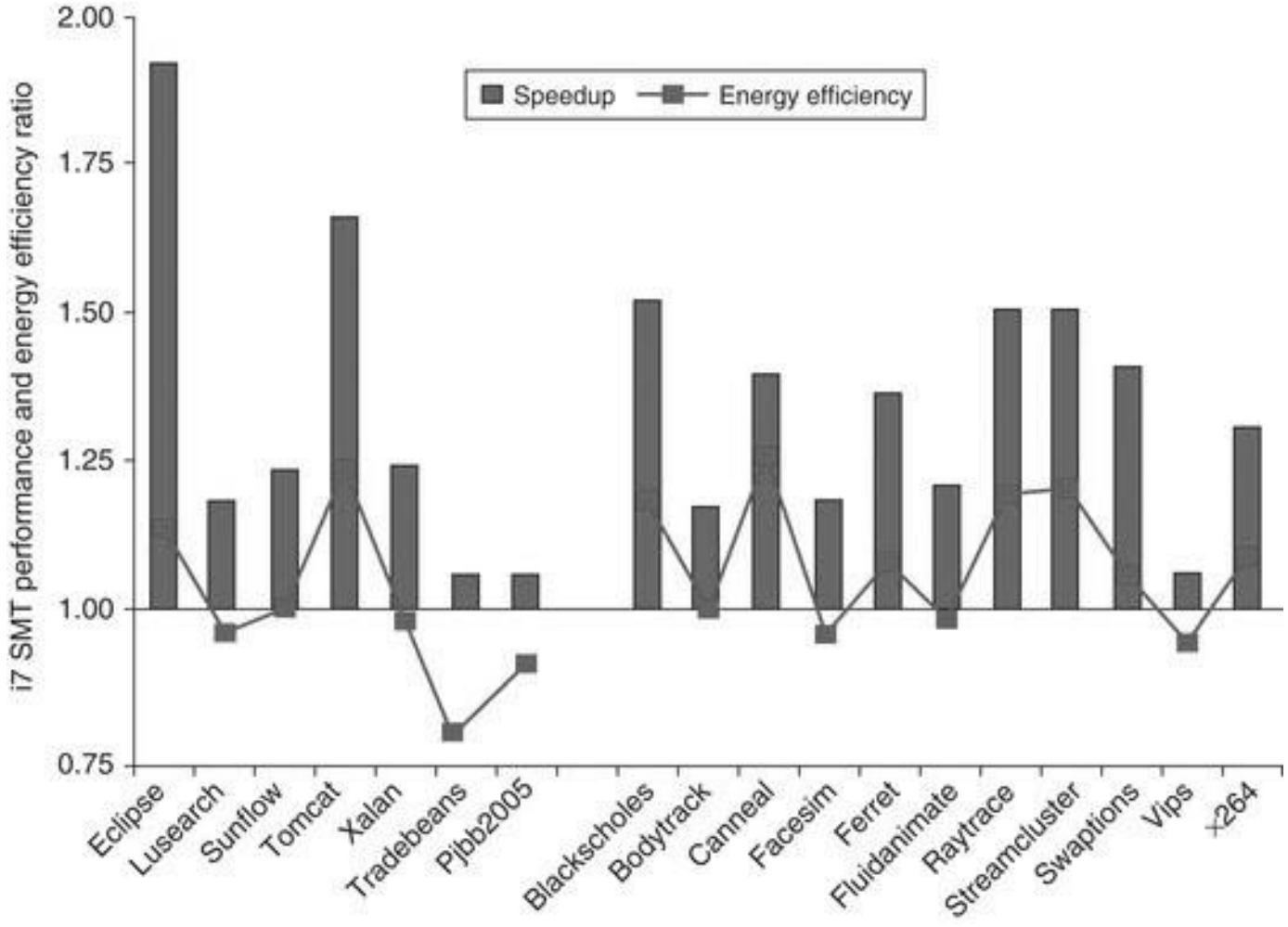
- **Fine-grain MT can only increase throughput by a factor of 2.1**
- **SMT has much higher speedup**
- **Alternatives to execute 4 instructions per cycle**
  - Four issue or full SMT with 3-4 threads
  - Dual issue SMT with 4 threads
  - Limited Connection SMT with 5 threads
  - Single issue SMT with 6 threads



Tullsen et al., Figure 3

SMT Performance: Java and PARSEC benchmarks

- **Figure shows speedup and energy efficiency (high is better) for Intel Core i7**
  - ➢Uses hyperthreading: similar to 2-way SMT
- **Speedup averages 1.28x for Java and 1.31x for PARSEC**
- **Energy Efficiency average 0.99 for Java and 1.07 for PARSEC**



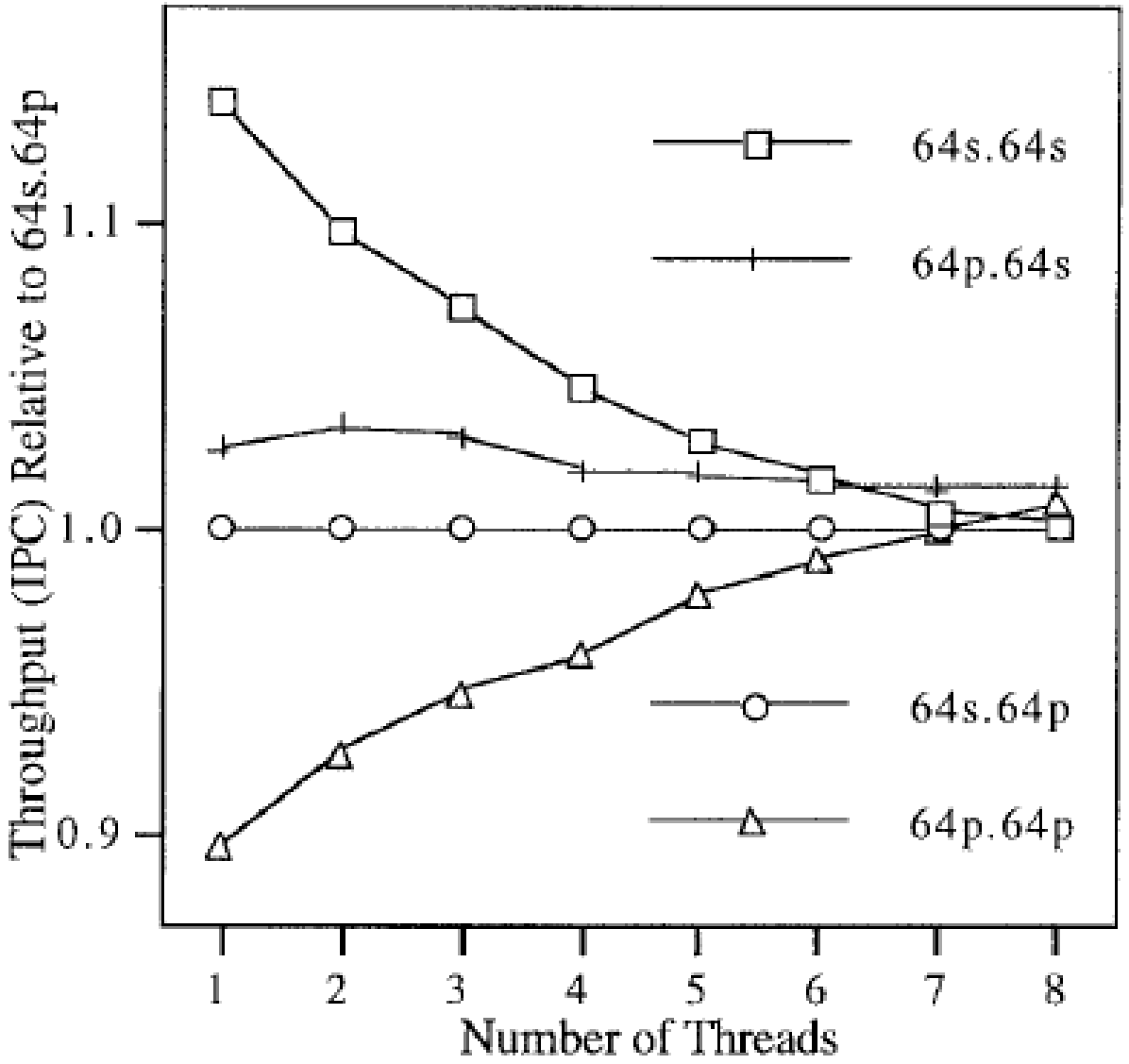
ARCH Figure 3.35

SMT Performance Side Effects

- **Lowest priority thread runs much slower than high priority thread**
- **Highest priority thread sees degraded performance as more threads are added**
  - Sharing of resources (e.g., caches, TLB, BP tables)
- **Caches are more strained by an MT workload vs. ST workload due to a decrease in locality**

Private vs. Shared Caches

- **Processor caches can be private (dedicated to specific thread) or shared (among all threads)**
- **When would shared caches perform better?**
  - Small number of threads
  - Significant instruction and data sharing
- **When would private caches perform better?**
  - Large number of threads
  - Minimal instruction and data sharing



Tullsen et al., Figure 4

SMT vs. Multiprocessors?

Purpose of Test	Common Elements	Specific Configuration	Throughput (instructions/cycle)
Unlimited FUs: equal total issue bandwidth, equal number of register sets (processors or threads)	**Test A**: FUs = 32 Issue bw = 8 Reg sets = 8	SM: 8 thread, 8-issue	6.64
		MP: 8 1-issue procs	5.13
	Test B: FUs = 16 Issue bw = 4 Reg sets = 4	SM: 4 thread, 4-issue	3.40
		MP: 4 1-issue procs	2.77
	Test C: FUs = 16 Issue bw = 8 Reg sets = 4	SM: 4 thread, 8-issue	4.15
		MP: 4 2-issue procs	3.44
Unlimited FUs: Test A, but limit SM to 10 FUs	**Test D**: Issue bw = 8 Reg sets = 8	SM: 8 thread, 8 issue, 10 FU	6.36
		MP: 8 1-issue procs, 32 FU	5.13
Unequal Issue BW: MP has up to four times the total issue bandwidth	**Test E**: FUs = 32 Reg sets = 8	SM: 8 thread, 8-issue	6.64
		MP: 8 4-issue procs	6.35
	Test F: FUs = 16 Reg sets = 4	SM: 4 thread, 8-issue	4.15
		MP: 4 4-issue procs	3.72
FU Utilization: equal FUs, equal issue bw, unequal reg sets	**Test G**: FUs = 8 Issue BW = 8	SM: 8 thread, 8-issue	5.30
		MP: 2 4-issue procs	1.94

Tullsen et al., Figure 5

SMT vs. Multiprocessors Discussion

- **SMT outperforms multiprocessing for all scenarios considered. Why?**
- **Advantages of SMT vs. MP**
  - Area efficiency
  - Reducing number of threads (i.e., threads becoming idle) allows other threads to progress faster in SMT processors, no change in MP
  - Granularity and flexibility of design: Unit of design is a whole processor for MP, more flexible in SMT
- **Disadvantages?**

SMT Design Issues

- **Hardware complexity**
  - Scheduling hardware requirements increase with threads
  - Register file size increase
  - May need more ports
- **Pipeline depth**
  - Bigger structures (e.g., register file) require longer access time
  - Leads to increasing the number of pipeline stages
- **Issue policy**
  - Fixed thread priority
  - Round-Robin priority
  - ICOUNT
  - Others?

Speculative Multithreading

Why Speculative Multithreading?

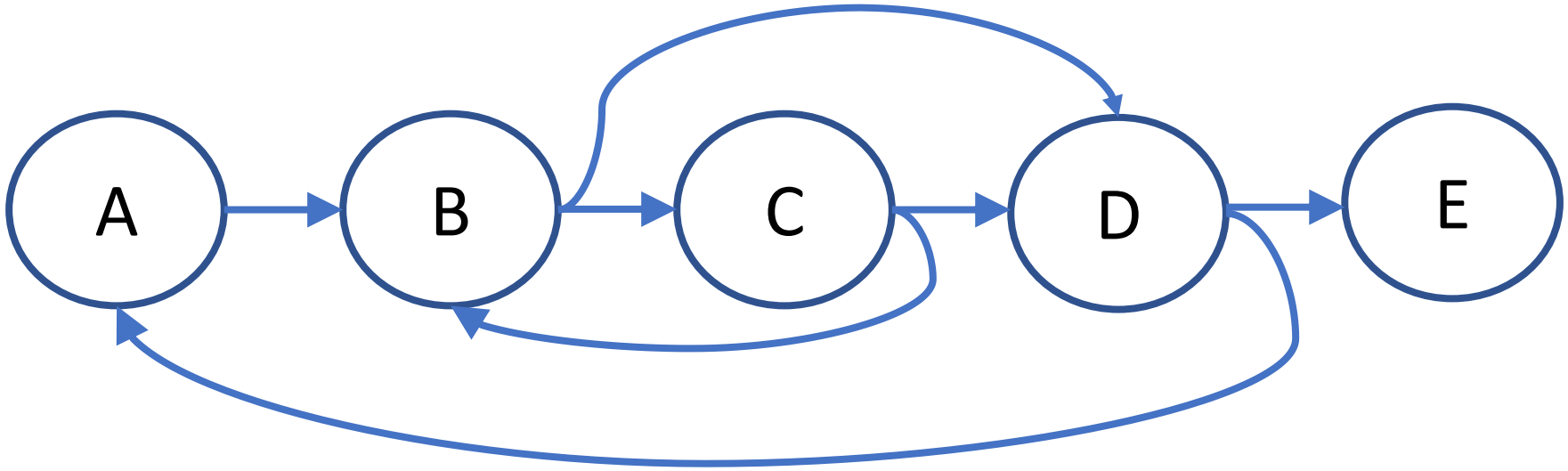
- **Multithreading helps performance when workloads have TLP**
- **What about single-threaded workloads?**
  - Traditional OoO superscalar processors can only consider parallelism within an instruction window
  - Even with large window sizes, many instructions are dependent and need to wait for other instructions to execute
- **Can we use multithreading to improve ST performance?**
  - Key idea: Split program into large tasks (with compiler help), issue tasks independently on different threads
  - If dependent tasks are correct, SpMT achieves significant performance improvement for ST workloads using multithreading execution resources
- **Speculative Multithreading (also called Thread-Level Speculation "TLS") uses speculation with multithreading to improve ST performance**

Speculative Multithreading (Thread-Level Speculation)

- **Sequential program is divided into tasks**
- **Each task may have a predecessor task and/or a successor task, defined by sequential program order**
- **First task (based on sequential program order) is non-speculative. Following tasks that are executed speculatively should not violate sequential execution model**
- **If violation of sequential model is detected, hardware needs to stop speculative tasks and restore state to sequential state**
- **Incorrect speculation is expensive: Wasted work (energy) with no performance gain**
  - Speculative tasks should be based on easy-to-predict branches

Hard-to-Predict vs. Easy-to-Predict Branches

- Consider a program's control flow graph:



- Hard-to-predict branches are included inside a task (thread), i.e., "intra-task branches"
- Easy-to-predict branches can be used to start new task than can be run simultaneously with previous tasks, i.e., "inter-task branches"

Example Program: Parallel Search

- **Program reads a symbol from a buffer then searches for it in a linked list**
  - If symbol is present, process it. Otherwise, add it to list
- **Easy to predict branch at the end of the "for (indx=0…)" loop, so can run multiple tasks in parallel**
- **Sometimes parallel executions can conflict (e.g., searching for symbol currently being added). So we need to go back to sequential execution**

```
for (indx = 0; indx < BUFSIZE; indx++) {
        /* get the symbol for which to search */
        symbol = SYMVAL(buffer[indx]);

        /* do a linear search for the symbol in the list */
        for (list = listhd; list; list = LNEXT(list)) {
                /* if symbol already present, process entry */
                if (symbol == LELE(list)) {
                        process(list);
                        break;
                }
        }

        /* if symbol not found in the list, add to the tail */
        if (!list) {
                addlist(symbol);
        }
}
```

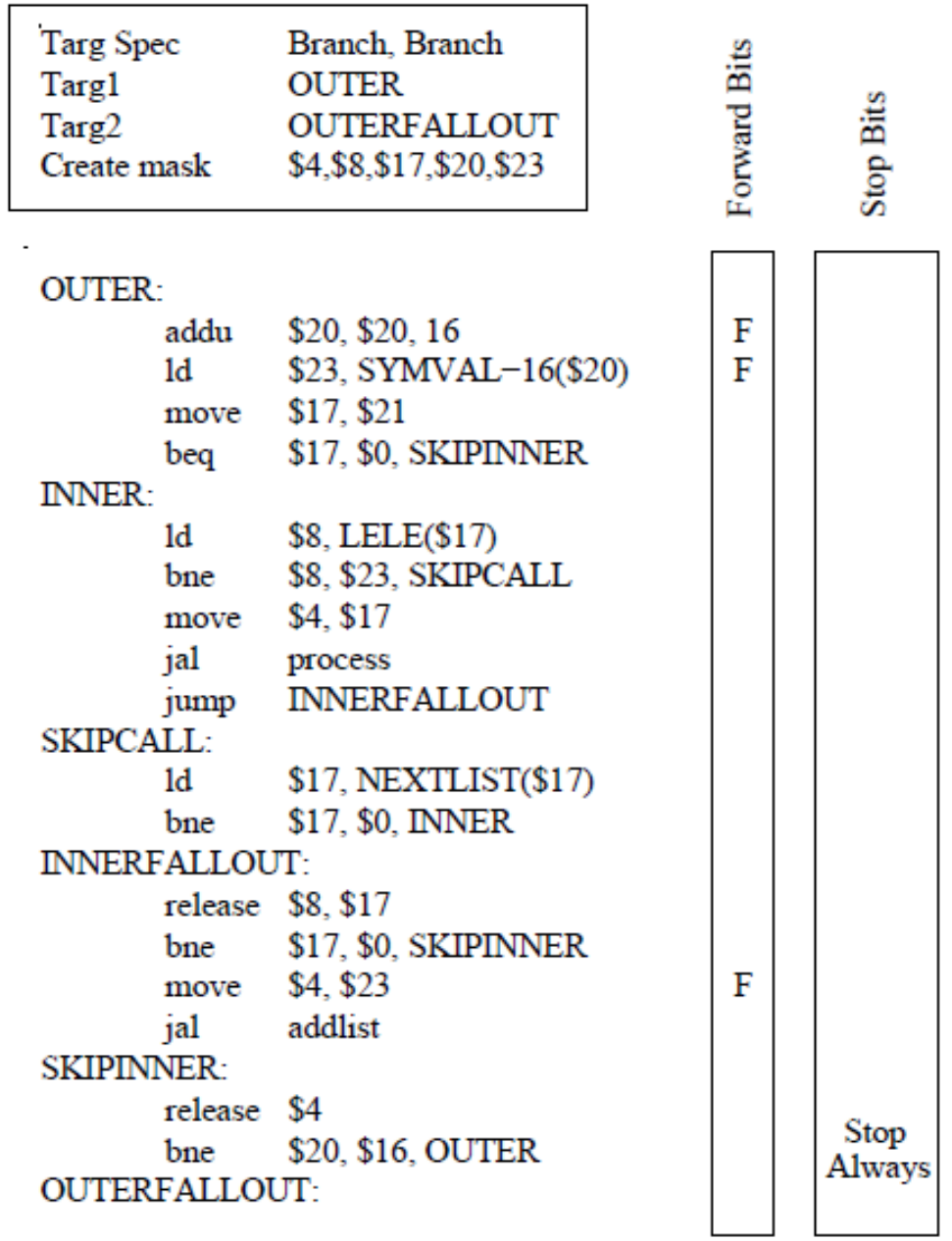
Sohi et al., Figure 3

Multiscalar Programs

- **Each task has to specify which registers it creates, how to forward register values, when the task ends, and which tasks follow it**
  - Information stored in *Task Descriptor*
- **Create Mask**
  - Register values that a task might produce
  - Conservative definition including all registers that could be produced (even if they are not produced in a particular instance of the task)
- **Forward Bits**
  - One bit associated with every instruction in task
  - Indicates whether destination register value is the last write by current task to that register, should be forwarded to subsequent tasks
- **Stop Bits**
  - Needs to check if conditions for stopping current task are satisfied at current instruction then task is exited
- **Release Instructions**
  - Indicates no further updates to register, can be forwarded to subsequent tasks

Multiscalar Programs: Example

- **Parallel search program**
- **Task has two possible targets: OUTER and OUTERFALLOUT**
- **Task can create up to 5 registers (4, 8, 17, 20, 23)**
  - Program indicates when task can forward a register and when it can release it



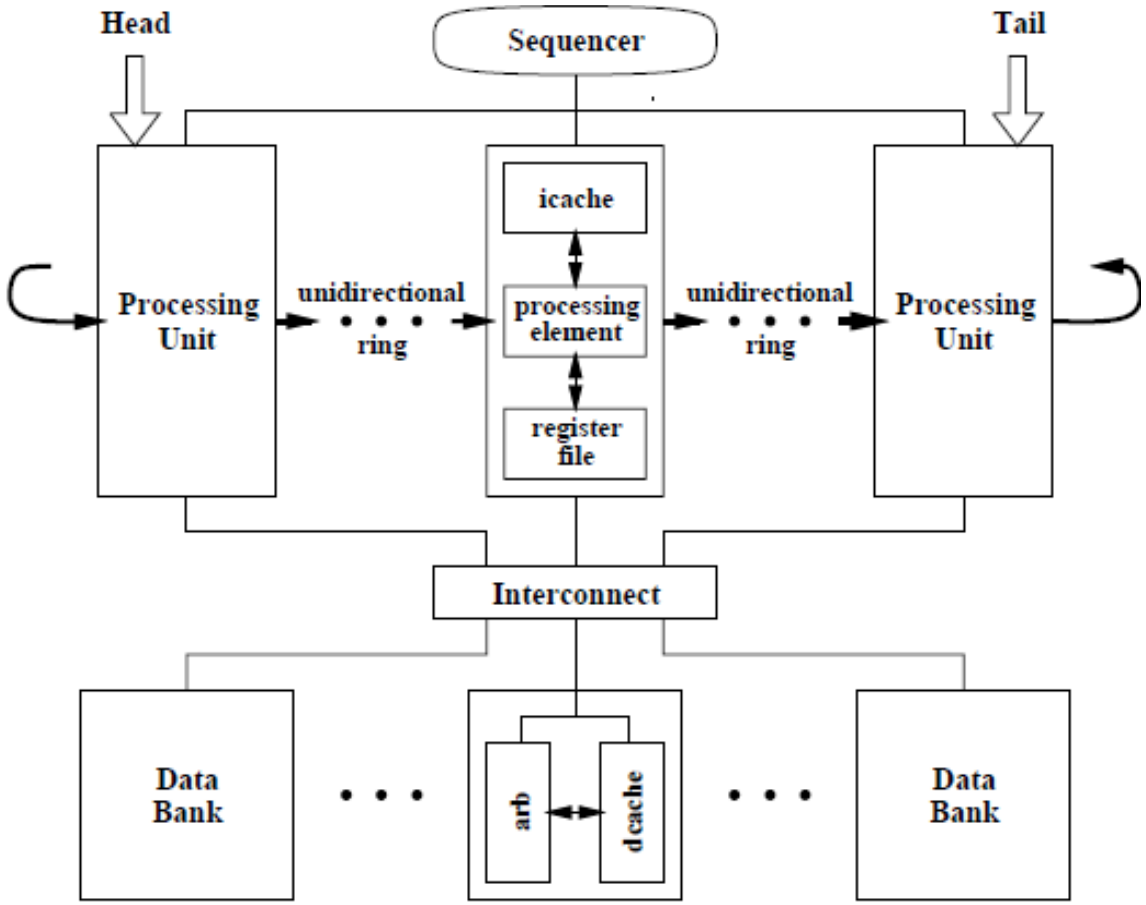
Targ Spec	Branch, Branch
Targ1	OUTER
Targ2	OUTERFALLOUT
Create mask	$4,$8,$17,$20,$23

		Forward Bits	Stop Bits
OUTER:			
addu	$20, $20, 16	F	
ld	$23, SYMVAL−16($20)	F	
move	$17, $21		
beq	$17, $0, SKIPINNER		
INNER:			
ld	$8, LELE($17)		
bne	$8, $23, SKIPCALL		
move	$4, $17		
jal	process		
jump	INNERFALLOUT		
SKIPCALL:			
ld	$17, NEXTLIST($17)		
bne	$17, $0, INNER		
INNERFALLOUT:			
release	$8, $17		
bne	$17, $0, SKIPINNER		
move	$4, $23	F	
jal	addlist		
SKIPINNER:			
release	$4		
bne	$20, $16, OUTER		Stop Always
OUTERFALLOUT:			

Sohi et al., Figure 4

Multiscalar Implementation

- Processing Units (PUs) execute tasks
- Each PU has a processing element, instruction cache and a register file
- Sequencer assigns tasks to PUs
- After task is assigned, PU fetches instructions and executes task until completion
- May need to use multi-version caches to store multiple versions of same value simultaneously
- At a high level, Multiscalar could also use multiple threads in an SMT processor



Sohi et al., Figure 1

SpMT/TLS Implementation Cost

- **When speculative tasks violate sequential order, they need to be squashed**
  - Consumes power without gaining performance
- **Need to support multi-version caches for store values written by speculative tasks**
  - Values can only be written to memory from non-speculative tasks
  - Adds complexity to cache design
- **Dependence checking across tasks may require complex hardware**
- **Requires compiler support: Program analysis, creating task descriptors, adding code for dependence checking**
- **Adds more instructions or prefix bits to existing instructions**
  - Increases program size
  - Code may not be portable across processor implementations
- **Some optimizations have been proposed to reduce energy overhead and hardware cost**

VLIW Architectures

Compiler Optimizations to Improve ILP

- **Problem: Short basic blocks (4-6 instructions on average) limit ILP**
  - Small pool of instructions, potentially dependent, to issue/execute in parallel
- **Loop Unrolling**
  - Used to increase number of instructions in a basic block
  - Can be combined with other compiler optimizations to improve performance
  - Adds more static instructions (program size increases)
  - Can reduce dynamic instruction count by combining or removing redundant instructions
- **Code motion**
  - Move instructions earlier to allow enough time for their dependent instructions to have operands available
  - Move instructions later to allow sources to be produced
  - Compiler needs knowledge of pipeline and latencies of different operations
  - OK if within basic block; need cleanup code if across branch boundary

Loop Unrolling: Example

Source Code:

```
for (i = 0; i < 100; i++)
        a[i] = a[i] + b[i];
```

Assembly:

```
        LD  R3, #100       ; loop index
LOOP:   LD.D F0, 0(R1)     ; array a[i] pointer
        LD.D F2, 0(R2)     ; array b[i] pointer
        ADD.D F4, F0, F2   ; add a[i] and b[i]
        S.D F4, 0(R1)      ; store a[i]
        ADD R1, R1, 8      ;move to next element in a
        ADD R2, R2, 8      ; move to next element in b
        SUB R3, R3, #1     ; decrement loop index
        BNE R3, #0, LOOP   ; go to start if index>0
```

- Loop adds corresponding elements of arrays a and b into array a
- Loop executes for 100 iterations
- Loop iterations are independent

Loop Unrolling: Example

Source Code:

```
for (i = 0; i < 100; i++)
        a[i] = a[i] + b[i];
```

Assembly:

```
        LD  R3, #100        ; loop index
LOOP:   LD.D F0, 0(R1)      ; array a[i] pointer
        LD.D F2, 0(R2)      ; array b[i] pointer
        ADD.D F4, F0, F2    ; add a[i] and b[i]
        S.D F4, 0(R1)       ; store a[i]
        ADD R1, R1, 8       ; move to a[i+1]
        ADD R2, R2, 8       ; move to b[i+1]
        SUB R3, R3, #1      ; decrement loop index
        BNE R3, #0, LOOP    ; go to start if index>0
```

After unrolling:

```
        LD  R3, #100
LOOP:   LD.D F0, 0(R1)
        LD.D F2, 0(R2)
        ADD.D F4, F0, F2
        S.D F4, 0(R1)
        ADD R1, R1, 8
        ADD R2, R2, 8
        SUB R3, R3, #1
        LD.D F0, 0(R1)
        LD.D F2, 0(R2)
        ADD.D F4, F0, F2
        S.D F4, 0(R1)
        ADD R1, R1, 8
        ADD R2, R2, 8
        SUB R3, R3, #1
        BNE R3, #0, LOOP
```

Loop Unrolling: Example

Source Code:

```
for (i = 0; i < 100; i++)
        a[i] = a[i] + b[i];
```

Assembly:

```
        LD  R3, #100        ; loop index
LOOP:   LD.D F0, 0(R1)      ; array a[i] pointer
        LD.D F2, 0(R2)      ; array b[i] pointer
        ADD.D F4, F0, F2    ; add a[i] and b[i]
        S.D F4, 0(R1)       ; store a[i]
        ADD R1, R1, 8       ; move to a[i+1]
        ADD R2, R2, 8       ; move to b[i+1]
        SUB R3, R3, #1      ; decrement loop index
        BNE R3, #0, LOOP    ; go to start if index>0
```

After removing redundant instructions:

```
        LD  R3, #100
LOOP:   LD.D F0, 0(R1)
        LD.D F2, 0(R2)
        ADD.D F4, F0, F2
        S.D F4, 0(R1)
        ADD R1, R1, 8
        ADD R2, R2, 8
        SUB R3, R3, #1
        LD.D F0, 8(R1)
        LD.D F2, 8(R2)
        ADD.D F4, F0, F2
        S.D F4, 8(R1)
        ADD R1, R1, 16
        ADD R2, R2, 16
        SUB R3, R3, #2
        BNE R3, #0, LOOP
```

Loop Unrolling: Example

Source Code:

```
for (i = 0; i < 100; i++)
        a[i] = a[i] + b[i];
```

Assembly:

```
        LD  R3, #100        ; loop index
LOOP:   LD.D F0, 0(R1)      ; array a[i] pointer
        LD.D F2, 0(R2)      ; array b[i] pointer
        ADD.D F4, F0, F2    ; add a[i] and b[i]
        S.D F4, 0(R1)       ; store a[i]
        ADD R1, R1, 8       ; move to a[i+1]
        ADD R2, R2, 8       ; move to b[i+1]
        SUB R3, R3, #1      ; decrement loop index
        BNE R3, #0, LOOP    ; go to start if index>0
```

After removing name dependences:

```
        LD  R3, #100
LOOP:   LD.D F0, 0(R1)
        LD.D F2, 0(R2)
        ADD.D F4, F0, F2
        S.D F4, 0(R1)
        LD.D F6, 8(R1)
        LD.D F8, 8(R2)
        ADD.D F10, F6, F8
        S.D F10, 8(R1)
        ADD R1, R1, 16
        ADD R2, R2, 16
        SUB R3, R3, #2
        BNE R3, #0, LOOP
```

Loop Unrolling: Example

Source Code:

```
for (i = 0; i < 100; i++)
        a[i] = a[i] + b[i];
```

Assembly:

```
        LD  R3, #100        ; loop index
LOOP:   LD.D F0, 0(R1)      ; array a[i] pointer
        LD.D F2, 0(R2)      ; array b[i] pointer
        ADD.D F4, F0, F2    ; add a[i] and b[i]
        S.D F4, 0(R1)       ; store a[i]
        ADD R1, R1, 8       ; move to a[i+1]
        ADD R2, R2, 8       ; move to b[i+1]
        SUB R3, R3, #1      ; decrement loop index
        BNE R3, #0, LOOP    ; go to start if index>0
```

After code motion:

```
        LD  R3, #100
LOOP:   LD.D F0, 0(R1)
        LD.D F2, 0(R2)
        LD.D F6, 8(R1)
        LD.D F8, 8(R2)
        ADD.D F4, F0, F2
        ADD.D F10, F6, F8
        S.D F4, 0(R1)
        S.D F10, 8(R1)
        ADD R1, R1, 16
        ADD R2, R2, 16
        SUB R3, R3, #2
        BNE R3, #0, LOOP
```

Loop Unrolling and Code Scheduling Basics

1. Determine which loops will be useful to unroll: Iterations are mostly independent except for loop maintenance code
2. Use different registers to avoid name dependences
3. Remove extra test and branch instructions
4. If loads and stores are independent, reorder to reduce dependences
5. Schedule code via code motion to reduce wait time for dependences (but still preserve original results)

Compiler Loop Unrolling: Pitfalls

- **Limited benefit from additional unrolling**
  - In previous example, stall cycles mostly removed with unrolling 4 times; further unrolling doesn't provide additional benefit
- **Code expansion**
  - Loop unrolling increases static program size, could lead to increase in instruction cache misses
  - May need "compensation code" when unrolling loops of unknown iteration count
- **Limited register file size**
  - Using different variables to avoid name dependences increases pressure on register file
  - Compiler register allocation problem gets more complex with additional variables
- **Complexities with multiprocessors and memory consistency models**
  - Code motion optimizations limited by memory consistency models
  - Need to consider multiple processors reading/writing shared data

Designing Multiple Issue Processors

- **Superscalar processors issue multiple (independent) instructions per cycle to improve ILP**
- **Options for multiple issue processors:**

1. **Statically-scheduled superscalar processors**
   - Scheduling is static using compiler optimizations, in-order execution
   - Can issue a variable number of instructions per cycle (based on dependences)
   - Simple hardware for dependence checking
2. **Dynamically-scheduled superscalar processors**
   - Hardware dynamically schedules independent instructions from instruction window (out-of-order)
   - Can issue a variable number of instructions per cycle (based on dependences)
   - Complex hardware needed for dependence checking and arbitration
3. **Very-Large Instruction Word (VLIW) processors**
   - Compiler schedules independent instructions as part of a large instruction word
   - Fixed number of available slots per instruction word

Comparing Multiple-Issue Processor Designs

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Cortex-A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

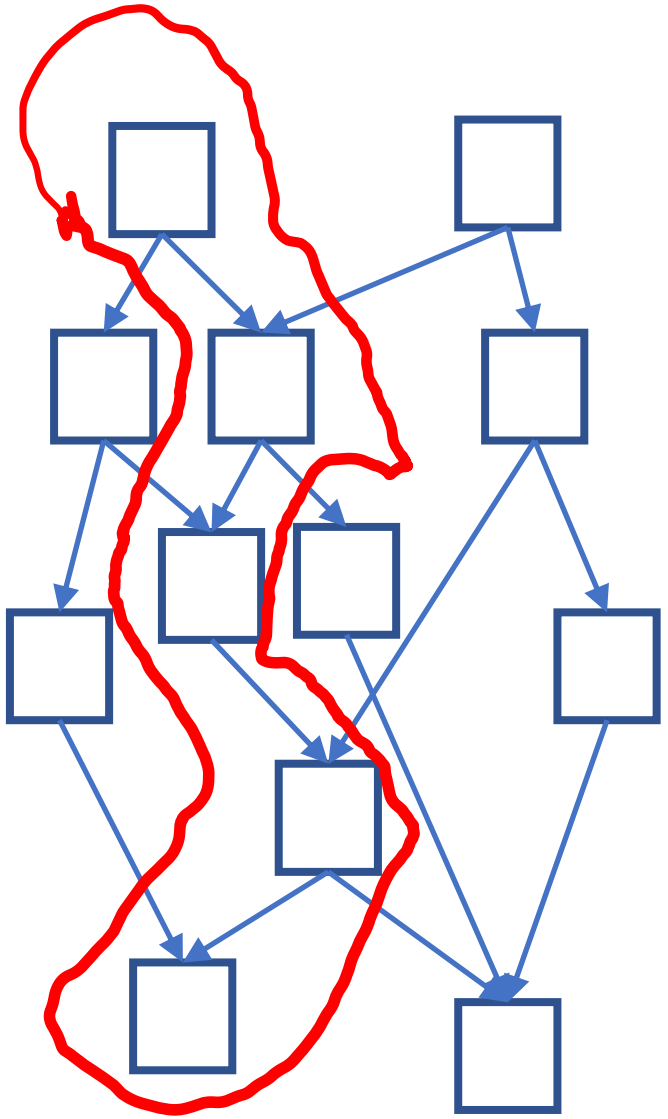
ARCH Figure 3.15

Very Long Instruction Word (VLIW) Architectures

- **Basic concept: Compiler extracts parallelism**
  - Static scheduling: Unroll loops, identify independent instructions
  - Trace scheduling: Next slide
- **Compiler output: Multiple operations to various execution units grouped together in one very long instruction**
- **Compiler schedules instructions based on pre-knowledge of hardware execution units and latencies**
- **Simple hardware**
  - No dependence checks needed in hardware
  - No need for arbitration before instruction issue

Trace Scheduling

- **Use profiling to identify most frequent traces in dynamic program**
- **Compiler schedules as if trace is one big basic block**
  - Moves instructions across branches
  - Moves loads above earlier stores
  - Compensation code for early exits from trace
  - NOPs inserted when slots cannot be filled with parallel operations
  - Compiler schedules for a specific hardware implementation
- **Repeat scheduling the next most frequent trace, including compensation code**



VLIW Example

- Start from unrolled loop in previous example
- Assume instruction word can include one FP operation, two memory operations and two integer/branch operation
- How would compiler schedule operations in an instruction word?

```
LD.D F0, 0(R1)
LD.D F2, 0(R2)
LD.D F6, 8(R1)
LD.D F8, 8(R2)
ADD.D F4, F0, F2
ADD.D F10, F6, F8
S.D F4, 0(R1)
S.D F10, 8(R1)
ADD R1, R1, 16
ADD R2, R2, 16
SUB R3, R3, #2
BNE R3, #0, LOOP
```

VLIW Example

```
LD.D F0, 0(R1)
LD.D F2, 0(R2)
LD.D F6, 8(R1)
LD.D F8, 8(R2)
ADD.D F4, F0, F2
ADD.D F10, F6, F8
S.D F4, 0(R1)
S.D F10, 8(R1)
ADD R1, R1, 16
ADD R2, R2, 16
SUB R3, R3, #2
BNE R3, #0, LOOP
```

FP	MEM1	MEM2	INT/BR 1	INT/BR 2
	LD.D F0, 0(R1)	LD.D F2, 0(R2)		
ADD.D F4,F0,F2	LD.D F6, 8(R1)	LD.D F8, 8(R2)		
ADD.D F10 F6,F8	S.D F4, 0(R1)			
	S.D F10, 8(R1)			
			ADD R1 R1,16	ADD R2,R2,16
			SUB R3,R3,#2	BNE R3,#0,LOOP

VLIW Issues (1): Code Expansion

- VLIW leads to much larger code size:

1. Filling up all instruction slots requires aggressive loop unrolling
2. Many issue slots are not used (filled with NOPs) which waste bits in instruction encoding
3. Adding compensation code further increases code size

- To reduce code expansion, programs could be compressed in memory and expanded when they get to instruction cache

VLIW Issues (2): Binary Compatibility

- **Compiled programs have fixed instruction word size**
- **Compiler requires knowledge about processor microarchitecture:**
  - Number of functional units of each type
  - Latency of each functional unit to determine when dependences are satisfied
  - Specific requirements for each unit (e.g., limitations on source registers)

- **Requires recompilation for new hardware**
  - Code need to be recompiled and migrated to new processor implementation
  - Recompilation needed if number of units is different, or if latencies of functional units are different
  - Compare to: superscalar processors

VLIW Issues (3): Dynamic Execution

- **Compiler cannot anticipate dynamic events or account for variable execution latencies**

1. **Cache misses**
   - Static scheduling assumes cache hits
   - VLIW requires blocking caches so a cache miss blocks issue for future instruction words, degrading performance
2. **Memory disambiguation**
   - Pointer references are assumed to be dependent, couldn't belong to same instruction word
   - Adds false dependences between loads and stores
3. **Branch outcomes**
   - Branch mispredictions lead to executing compensation code

VLIW Discussion

- **VLIW vs. vector architectures**
  - Vector architectures preferred for vector operations (such as the code example discussed earlier)
  - VLIW and other multiple-issue superscalar approaches preferred for general code since they can extract parallelism from less structured code
- **VLIWs need clever Jump mechanisms if instruction word supports multiple branches**
  - N tests, N+1 jump destinations
  - Similar to C's switch statement
- **VLIW compilers need to predict memory properties to determine when a load comes back from memory and schedule dependent instructions**
  - Example: cache hits and misses at different cache levels, memory bank information
  - Very difficult for compiler to predict dynamic behavior

EPIC Architectures

- **EPIC: Explicitly Parallel Instruction Computing**
- **Solves some of the issues in VLIW**
  - ➢Compiler provides register dependence information and hardware schedules accordingly
    - ❑Eliminates NOPs
    - ❑Provides backward compatibility
  - ➢Provides hardware mechanisms for events that cannot be predicted by the compiler
    - ❑Predication for branches
    - ❑Control speculation
    - ❑Data speculation

The End?

What We Covered In this Course

- Superscalar Processors: OoO execution, dynamic scheduling, issue logic
- Speculative Execution: Branch prediction, memory dependence prediction
- Technology: Trends, impact on architecture, power and energy
- Domain Specific Accelerators, Dataflow, SIMD, Vector Processors
- Memory Hierarchy: Caches, memory-level parallelism, cache prefetching, replacement and insertion policies, DRAM basics, novel memory technologies
- Parallel Architectures: Multicore processors, shared-memory and distributed memory architecture
- Cache Coherence Protocols and Memory Consistency Models
- Multithreading: SMT, SpMT, VLIW architectures

Other Important Topics Not Covered In This Course

- Graphics Processors: GPUs, GPGPUs
- Dataflow architectures
- Security
- Reliability
- Virtual Memory implementations
- On-chip interconnection networks (Networks-on-Chip “NoC”)
- Synchronization primitives and lock/barrier implementations
- Architecting warehouse-scale computers
- Embedded Processors
- … and many other topics

Reading Assignments

- ARCH Chapter 3.2, 3.7, 3.12 (Read)
- D. Tullsen et al., “Simultaneous Multithreading: Maximizing On-Chip Parallelism,” ISCA 1995 (Read)
- G. Sohi et al., “Multiscalar Processors” (Read)
- J. Renau et al., “Thread-Level Speculation on a CMP Can be Energy Efficient,” ICS, 2005 (Skim)