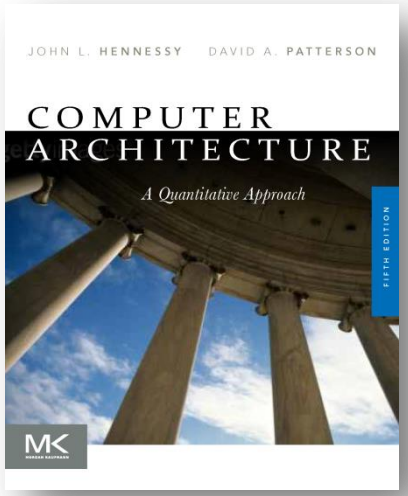


Chapter 3

Instruction-Level Parallelism and Its Exploitation

Adapted by Prof. Gheith Abandah

Contents

Contents

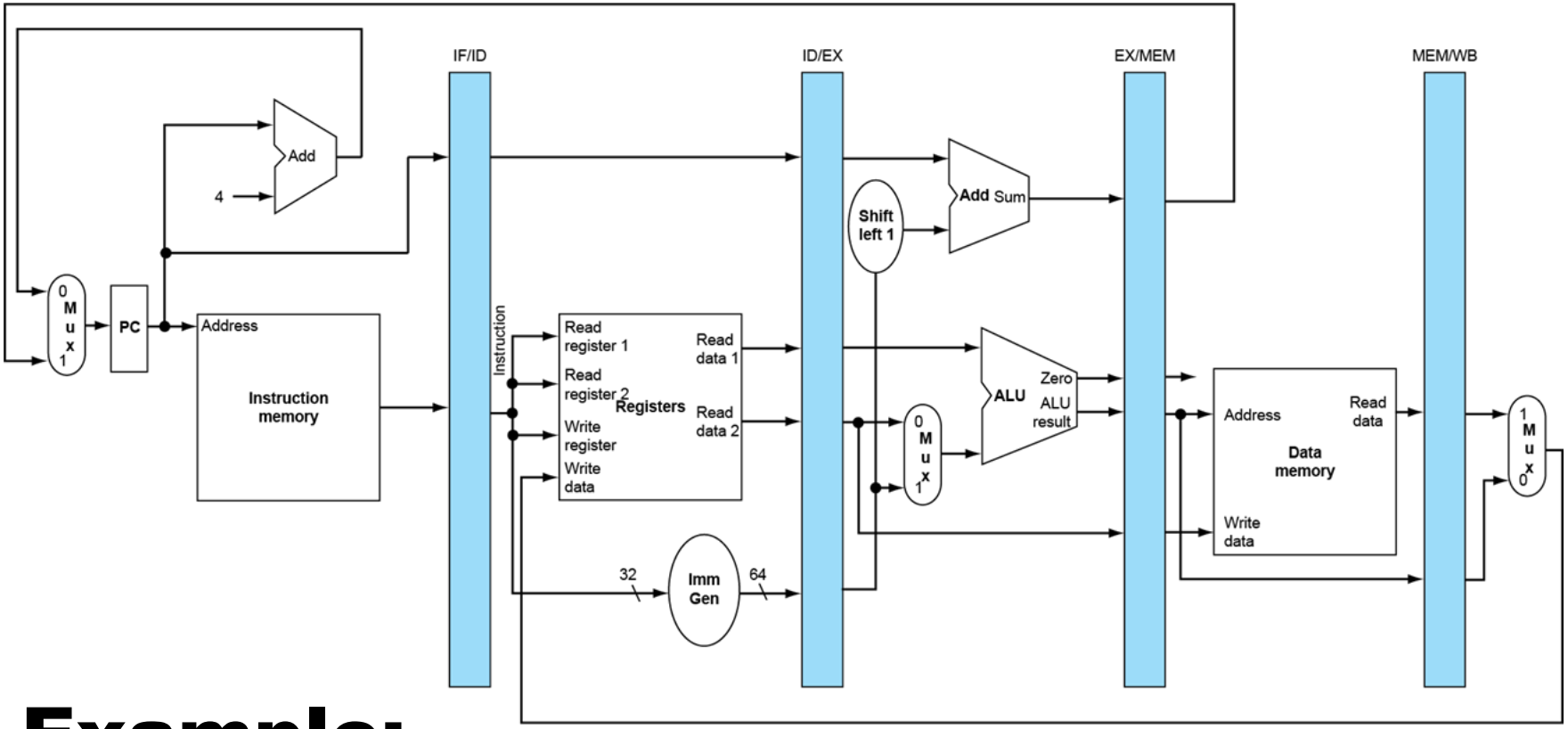
- Instruction-Level Parallelism: Concepts and Challenges
  - Introduction
  - Pipelining
  - ILP Approaches
  - ILP Challenges
  - Data Dependence
  - Name Dependence
  - Control Dependence

Introduction

- Instruction-Level Parallelism (ILP) allows overlapping the execution of instructions and reduce the execution time.

- Pipelining become universal technique in 1985
  - Overlaps execution of instructions
  - Exploits “Instruction Level Parallelism”

RISC-V Five-Stage Pipeline



Example:

```
ld   x22,16(x21)
add  x5,x6,x7
sub  x21,x21,x22
```

ILP Approaches

- Beyond pipelining, there are two main approaches:
  - Hardware-based dynamic approaches
    - Used in server and desktop processors
    - Not used as extensively in PMP processors
  - Compiler-based static approaches
    - Not as successful outside of scientific applications

ILP Challenges

- When exploiting instruction-level parallelism, goal is to minimize CPI
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls
- Parallelism with basic block is limited
  - Typical size of basic block = 3-6 instructions
  - Must optimize across branches
- Loop-Level Parallelism
  - Unroll loop statically or dynamically

Data Dependence

- Data dependency (Read-after-write, RAW)
  - Instruction *j* is data dependent on instruction *i* if
    - Instruction *i* produces a result that may be used by instruction *j*
    - Instruction *j* is data dependent on instruction *k* and instruction *k* is data dependent on instruction *i*
- Dependent instructions cannot be executed simultaneously

Data Dependence

- Dependencies are a property of programs
- Pipeline organization determines if dependence is detected and if it causes a stall

Example:

```
ld    x22,16(x21)
add   x5,x6,x22
```

- Data dependence conveys:
  - Possibility of a hazard
  - Order in which results must be calculated
  - Upper bound on exploitable instruction level parallelism
- Dependencies that flow through memory locations are difficult to detect

Name Dependence

- Two instructions use the same name but no flow of information.
- Not a true data dependence, but is a problem when reordering instructions.
- Two types:

1) **Antidependence** (WAR): instruction *j* writes a register or memory location that instruction *i* reads.
   - Initial ordering (*i* before *j*) must be preserved.

Example:

```
ld    x22,16(x21)
add   x21,x6,x7
```

Name Dependence

2) **Output dependence** (WAW): instruction *i* and instruction *j* write the same register or memory location.
   - Ordering must be preserved.

Example:

```
ld    x22,16(x21)
add   x22,x6,x7
```

- To resolve, use register renaming techniques.

Example:

```
ld    x22,16(x21)
add   x23,x6,x7
```

- Violating the order causes RAW, WAR, or WAW hazard.

Control Dependence

- Ordering of instruction *i* with respect to a branch instruction
  - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch
  - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch

Example 1:

```
   add x1,x2,x3
   beq x4,x0,L
   sub x1,x1,x6
L: …
   or x7,x1,x8
```

Example 2:

```
   add x1,x2,x3
   beq x12,x0,skip
   sub x4,x5,x6
   add x5,x4,x9
skip:
   or x7,x8,x9
```

Contents

Contents

- Basic Compiler Techniques for Exposing ILP
  - Pipeline Scheduling
  - Loop Unrolling

Pipeline Scheduling

- Separate dependent instruction from the source instruction by the pipeline latency of the source instruction
- Example:

```
for (i=999; i>=0; i=i-1)
  x[i] = x[i] + s;
```

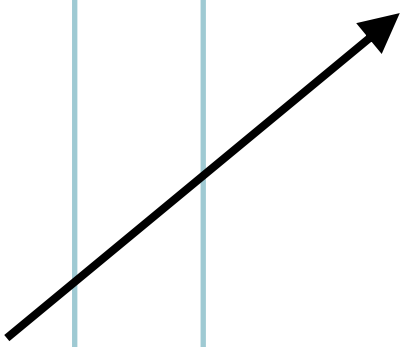
4 cycles in FP Unit

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Pipeline Scheduling

```
Loop: fld     f0,0(x1)
      fadd.d  f4,f0,f2
      fsd     f4,0(x1)
      addi    x1,x1,-8
      bne     x1,x2,Loop
```

```
Loop: fld     f0,0(x1)
      stall
      fadd.d  f4,f0,f2
      stall
      stall
      fsd     f4,0(x1)
      addi    x1,x1,-8
      bne     x1,x2,Loop
```



```
Loop: fld     f0,0(x1)
      addi    x1,x1,-8
      fadd.d  f4,f0,f2
      stall
      stall
      fsd     f4,0(x1)
      bne     x1,x2,Loop
```

Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried anti-dependencies and out put dependencies.
- Steps:
  1. Replicate the loop instructions n times
  2. Remove unneeded loop overhead
  3. Modify instructions
  4. Rename registers
  5. Schedule instructions

Loop Unrolling

- Unroll by a factor of 4 (assume # elements is divisible by 4)
- Eliminate unnecessary instructions, modify instructions, and rename registers.

```
Loop:   fld     f0,0(x1)
        fadd.d  f4,f0,f2
        fsd     f4,0(x1)     //drop addi & bne
        fld     f6,-8(x1)
        fadd.d  f8,f6,f2
        fsd     f8,-8(x1)    //drop addi & bne
        fld     f10,-16(x1)
        fadd.d  f12,f10,f2
        fsd     f12,-16(x1) //drop addi & bne
        fld     f14,-24(x1)
        fadd.d  f16,f14,f2
        fsd     f16,-24(x1)
        addi    x1,x1,-32
        bne     x1,x2,Loop
```

- Note: number of live registers vs. original loop

Loop Unrolling/Pipeline Scheduling

- Pipeline schedule the unrolled loop:

```
Loop:   fld     f0,0(x1)
        fld     f6,-8(x1)
        fld     f10,-16(x1)
        fld     f14,-24(x1)
        fadd.d  f4,f0,f2
        fadd.d  f8,f6,f2
        fadd.d  f12,f10,f2
        fadd.d  f16,f14,f2
        fsd     f4,0(x1)
        fsd     f8,-8(x1)
        fsd     f12,-16(x1)
        addi    x1,x1,-32
        fsd     f16,-24(x1)
        bne     x1,x2,Loop
```

- 14 cycles
- 3.5 cycles per element

Strip Mining

- Unknown number of loop iterations?
  - Number of iterations = *n*
  - Goal: Make *k* copies of the loop body
  - Generate pair of loops:
    - First executes *n* mod *k* times
    - Second executes *n* / *k* times
    - “Strip mining”

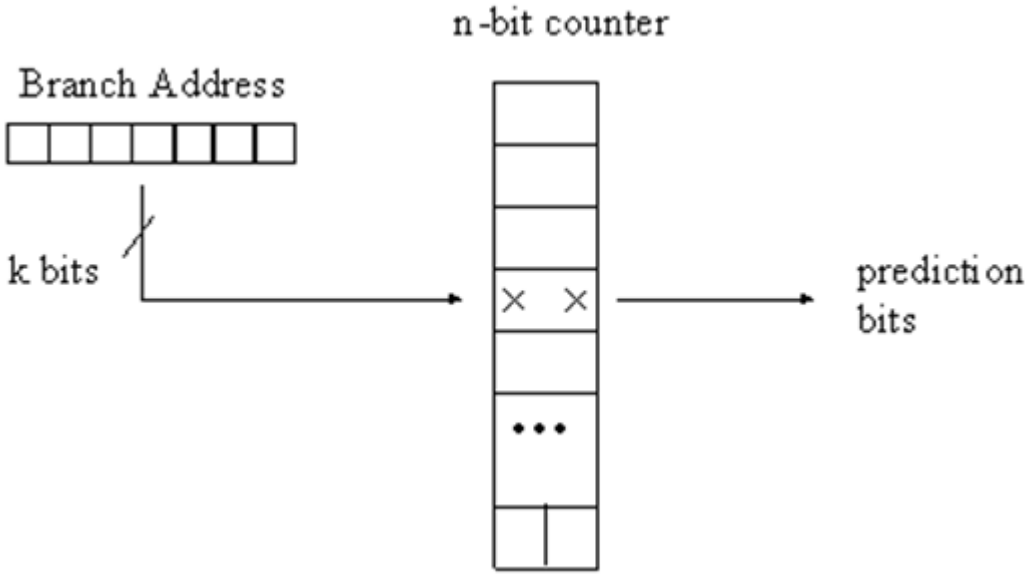
Contents

Contents

- Reducing Branch Costs with Advanced Branch Prediction
  - Basic 2-Bit Predictor
  - Correlating Predictor
  - Correlating gshare Predictor
  - Tournament Predictor
  - Tagged Hybrid Predictors

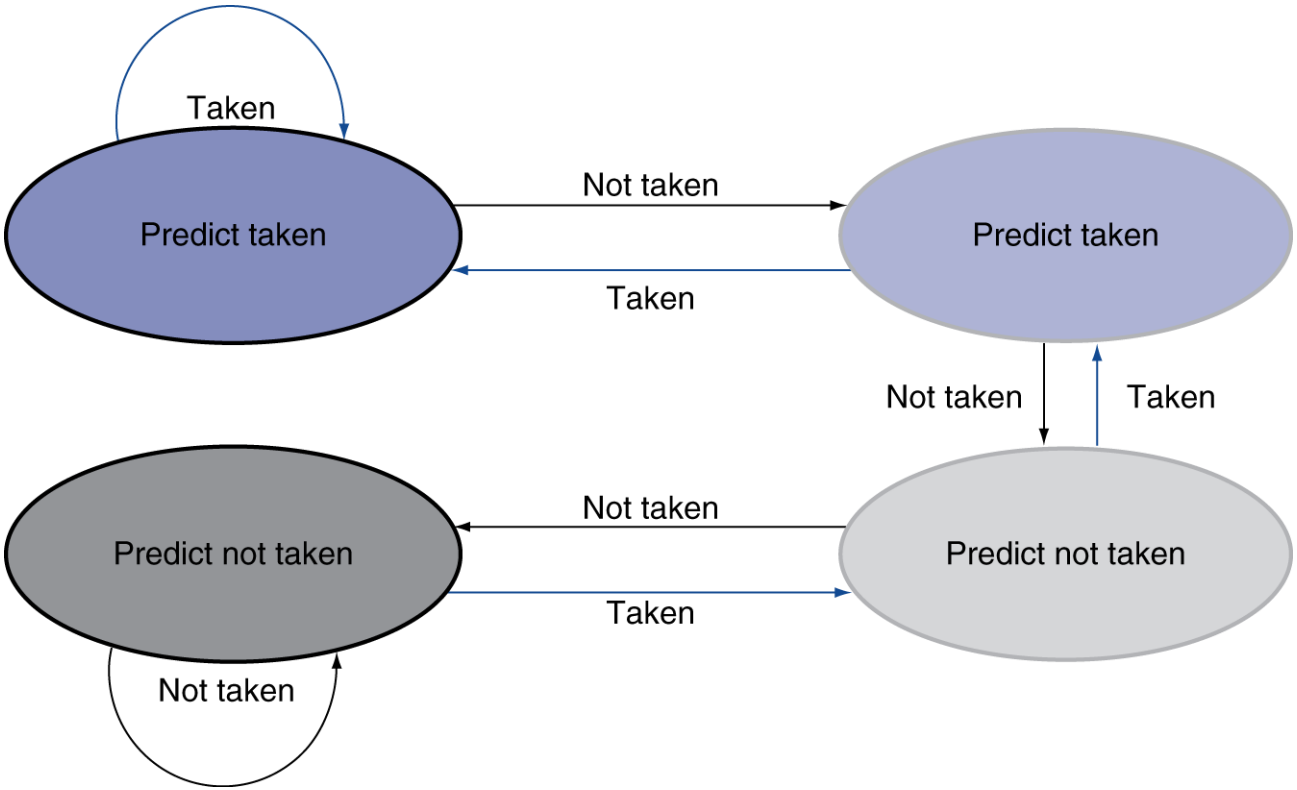
Basic 2-Bit Predictor

- Uses Branch History Table (BHT) with n = 2
- Table size = n × 2^k bits
- For each branch, predict taken or not taken



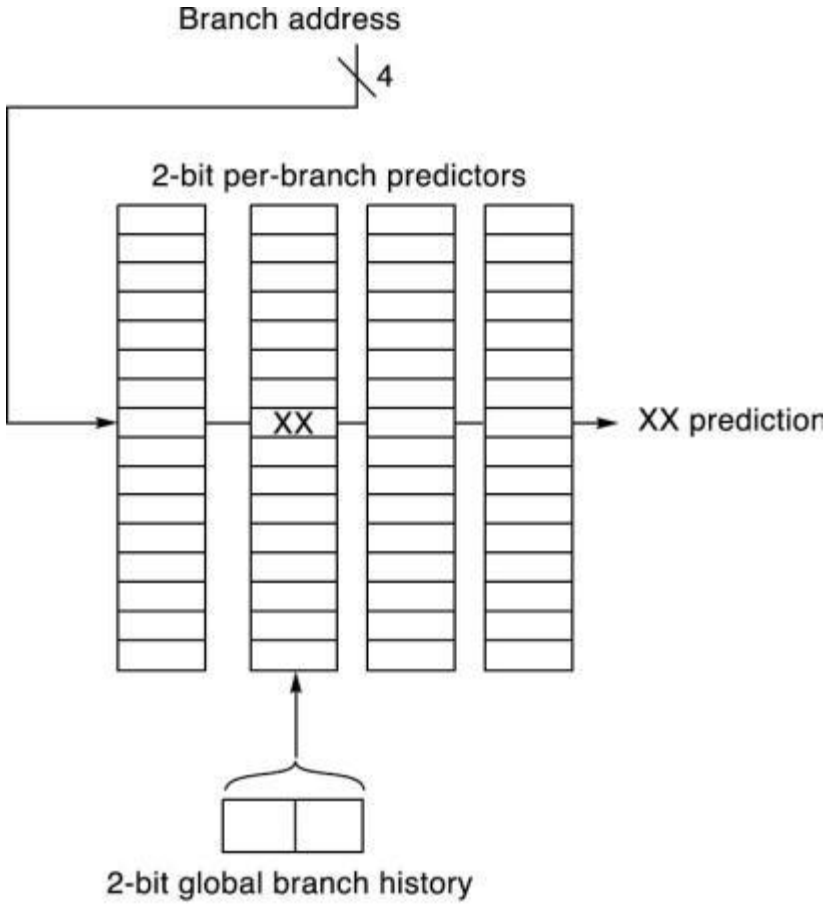
Basic 2-Bit Predictor

- Only change prediction on two successive mispredictions

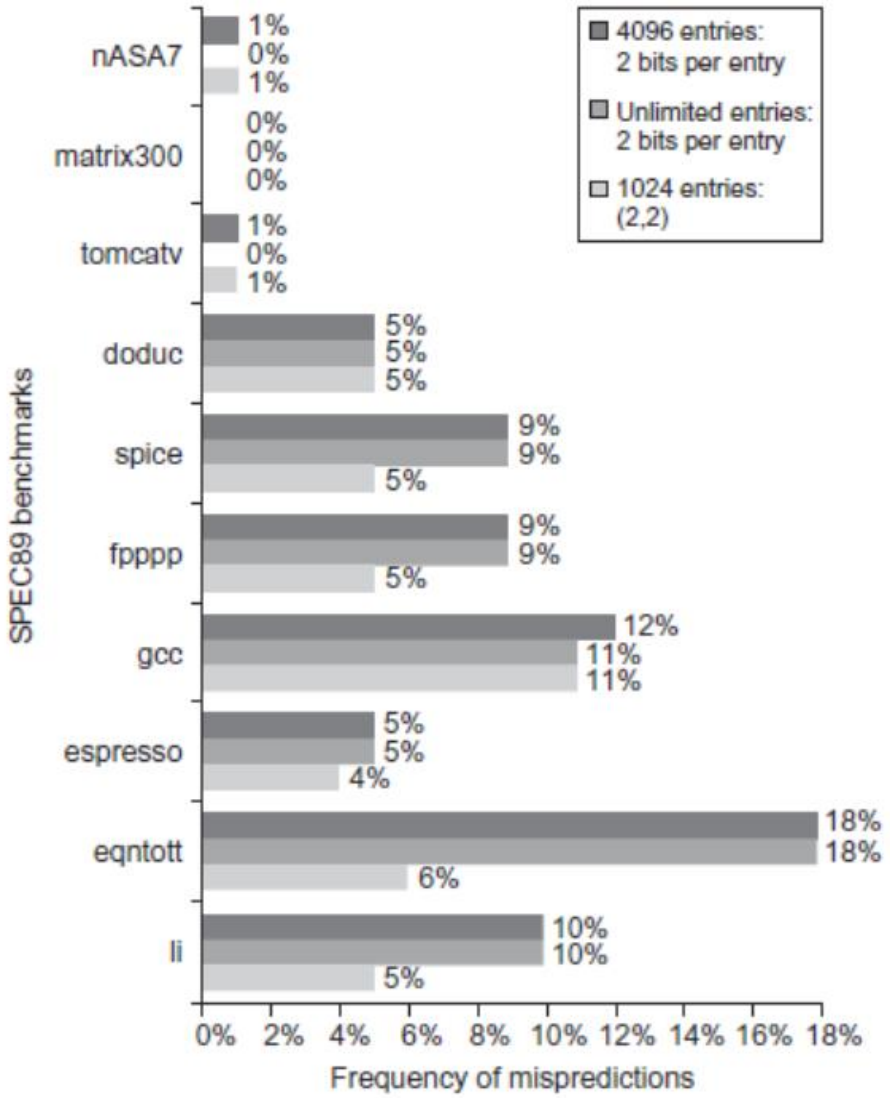


Correlating Predictor

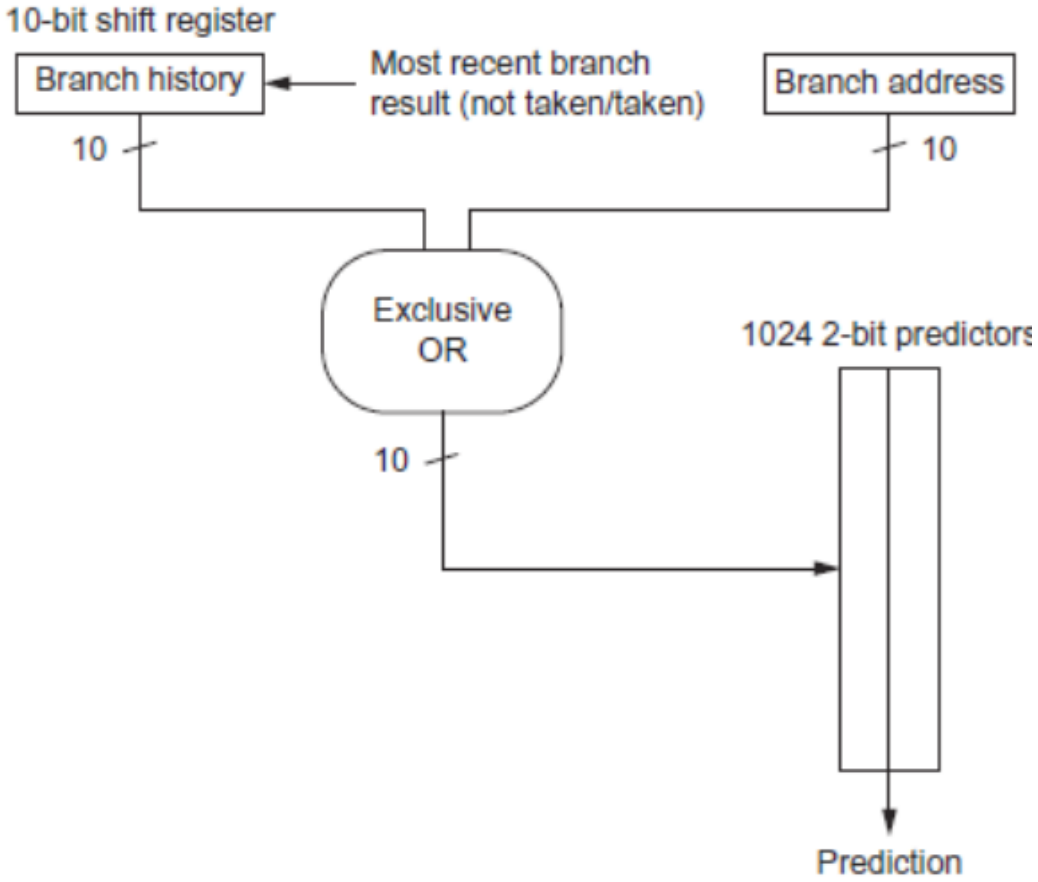
- Multiple 2-bit predictors for each branch
- One for each possible combination of outcomes of preceding *m* branches
  - (*m*,*n*) predictor: behavior from last *m* branches to choose from 2^m *n*-bit predictors
  - Size = n × 2^k × 2^m bits



Basic vs Correlating Predictors

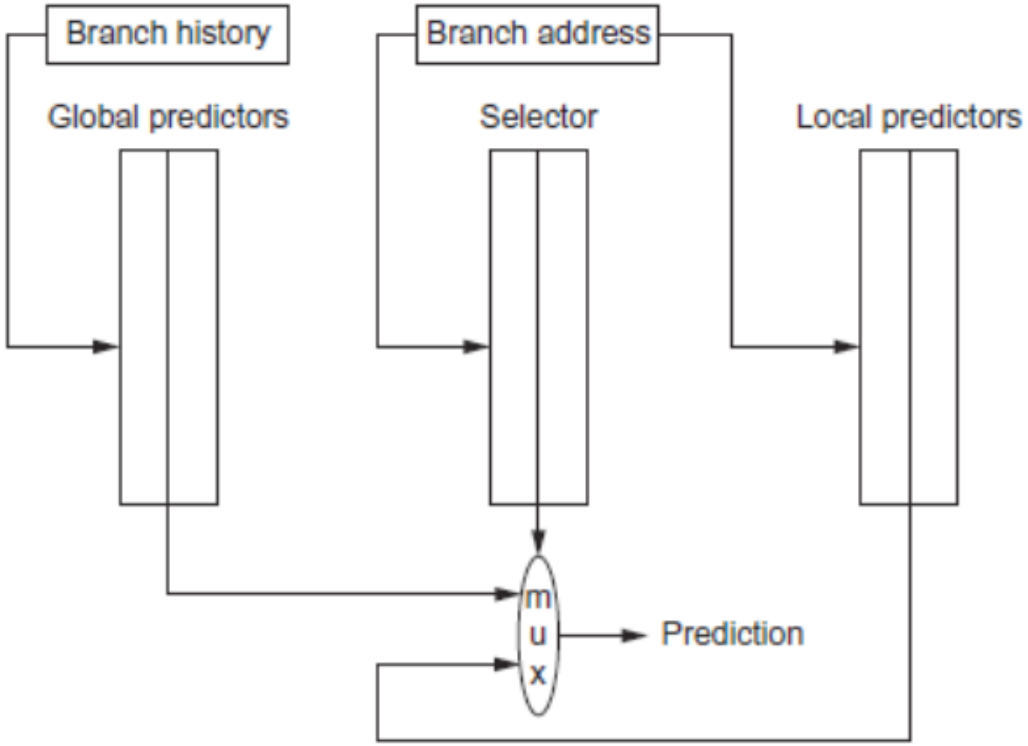


Correlating gshare Predictor

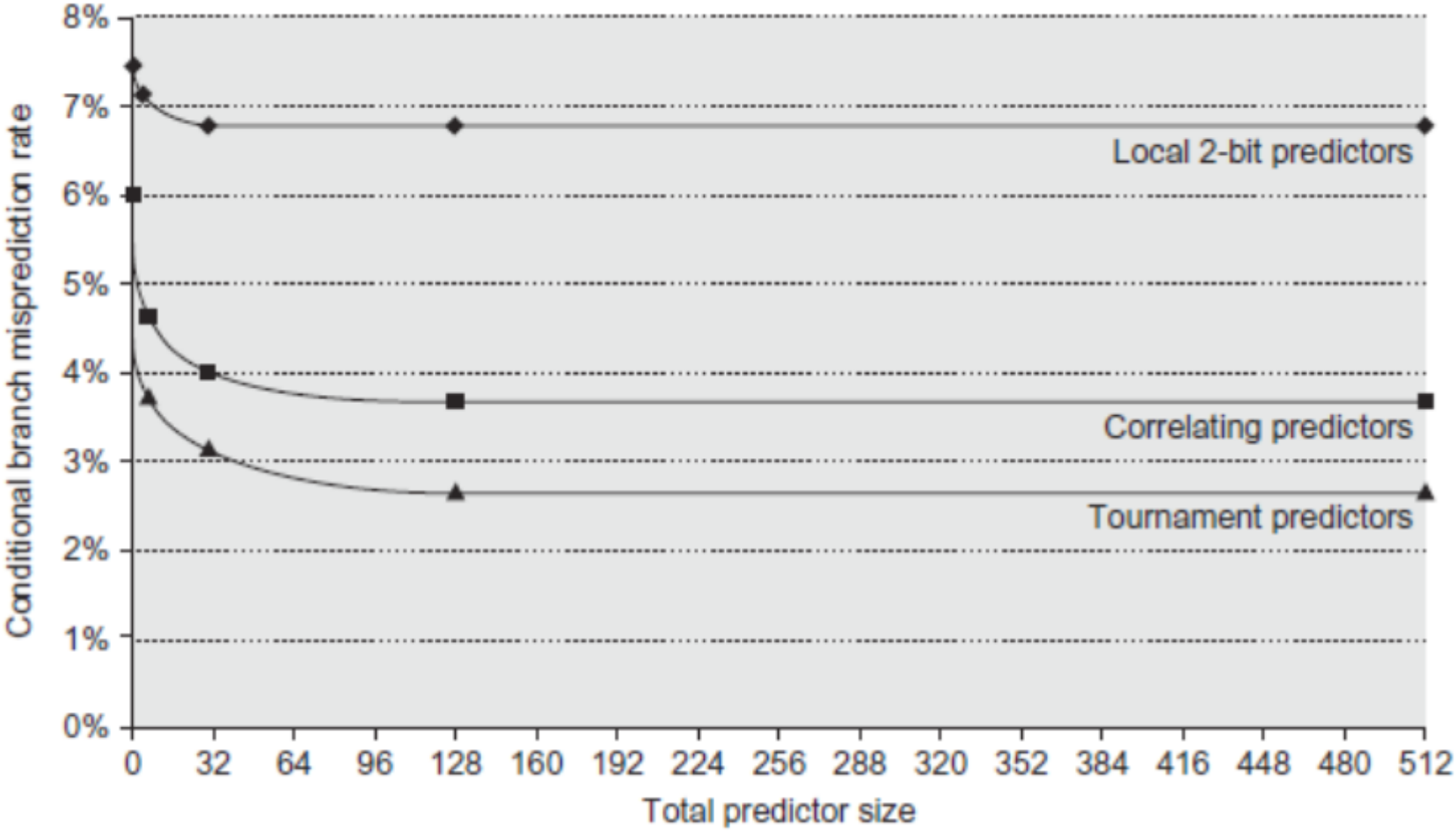


Tournament Predictor

- Combine correlating predictor with local predictor



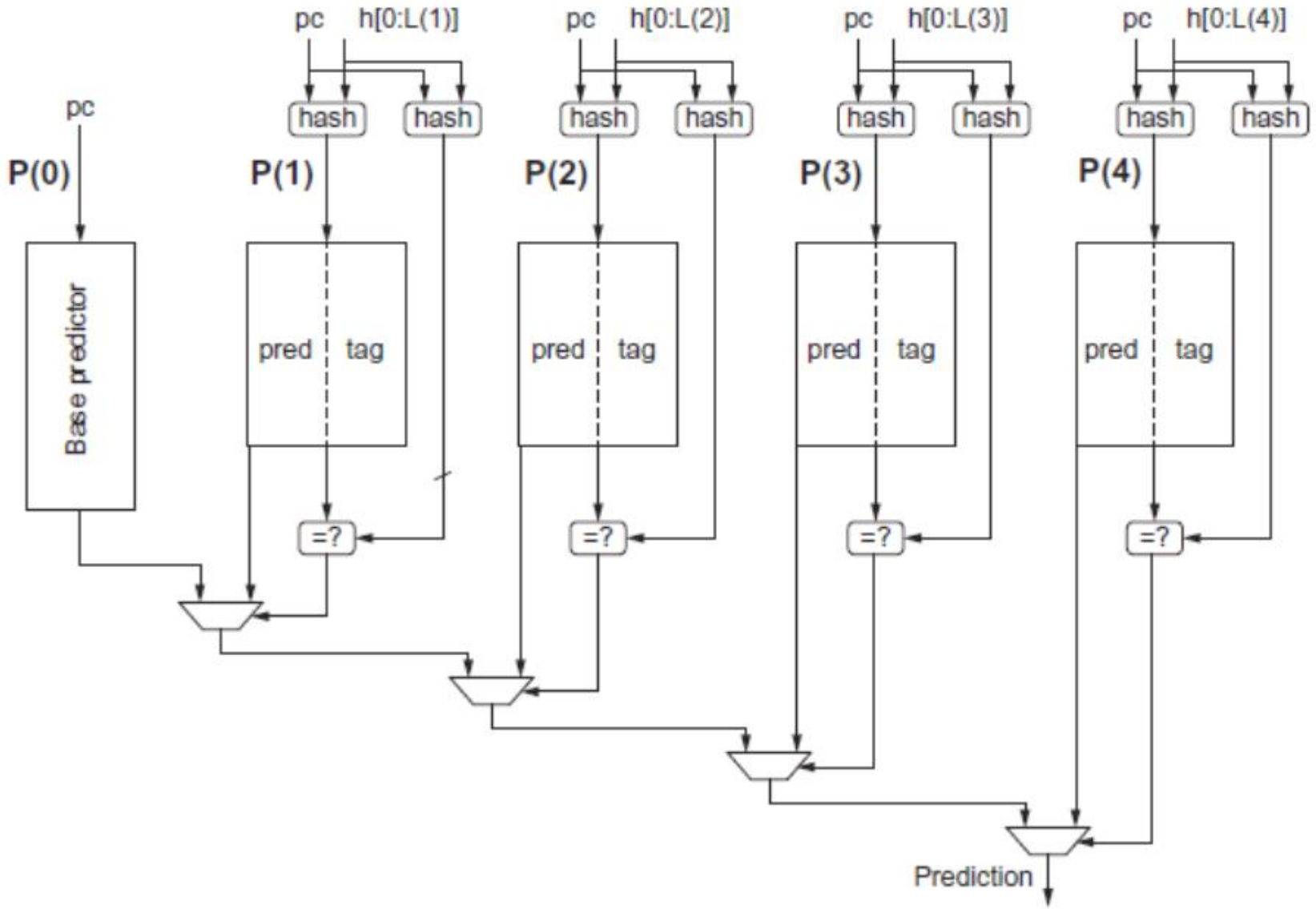
Local vs. Correlating vs. Tournament



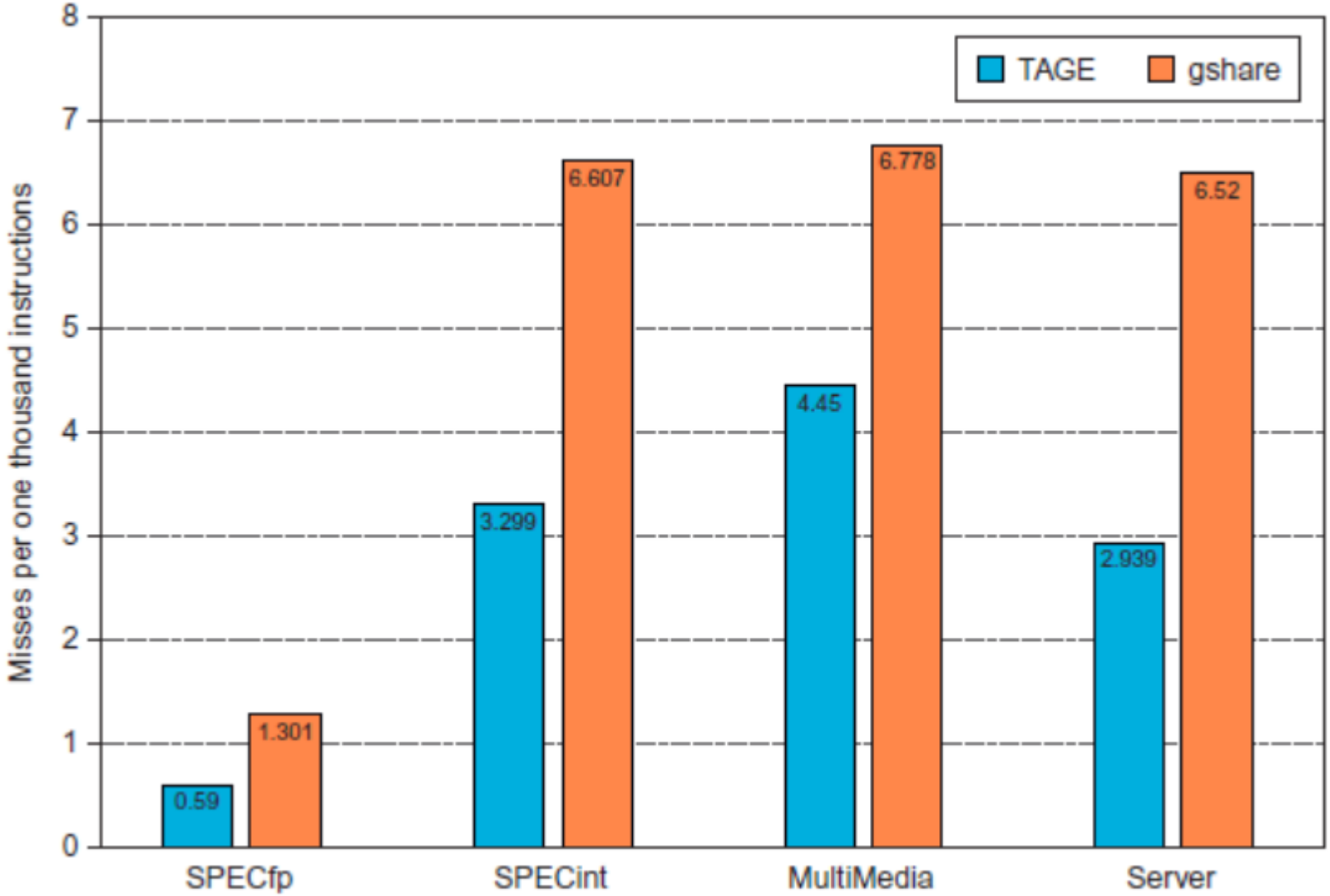
Tagged Hybrid Predictors

- Need to have predictor for each branch and history
  - Problem: this implies huge tables
  - Solution:
    - Use hash tables, whose hash value is based on branch address and branch history
    - Longer histories may lead to increased chance of hash collision, so use multiple tables with increasingly shorter histories

Tagged Hybrid Predictors



Tagged Hybrid Predictors



Contents

- Instruction-Level Parallelism: Concepts and Challenges
- Basic Compiler Techniques for Exposing ILP
- Reducing Branch Costs with Advanced Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- Dynamic Scheduling: Examples and Algorithms
- Hardware-Based Speculation
- Exploiting ILP Using Multiple Issue and Static Scheduling
- Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation
- Advanced Techniques for Instruction Delivery and Speculation
- Multithreading: Exploiting Thread-Level Parallelism to Improve Uniprocessor Throughput
- The Intel i7 6700 and ARM Cortex-A53
- Fallacies and Pitfalls

Contents

- Overcoming Data Hazards with Dynamic Scheduling
  - Introduction
  - Register Renaming
  - Tomasulo's Algorithm
  - Tomasulo's Algorithm Stages

Introduction

- Rearrange order of instructions to reduce stalls while maintaining data flow
- Advantages:
  - Compiler doesn't need to have knowledge of microarchitecture
  - Handles cases where dependencies are unknown at compile time
- Disadvantage:
  - Substantial increase in hardware complexity
  - Complicates exceptions

Introduction

- Dynamic scheduling implies:
  - Out-of-order execution
  - Out-of-order completion

- Example 1:

```
fdiv.d f0,f2,f4
fadd.d f10,f0,f8
fsub.d f12,f8,f14
```

  - `fsub.d` is not dependent, issue before `fadd.d`

Introduction

- Example 2:

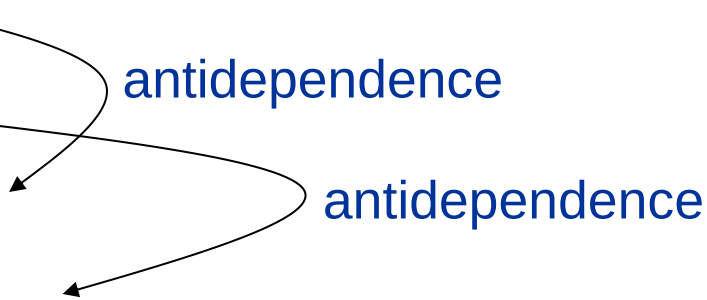
```
fdiv.d f0,f2,f4
fmul.d f6,f0,f8
fadd.d f0,f10,f14
```

  - `fadd.d` is not dependent, but the antidependence makes it impossible to issue earlier without register renaming

Register Renaming

- Example 3:

```
fdiv.d f0,f2,f4
fadd.d f6,f0,f8
fsd    f6,0(x1)
fsub.d f8,f10,f14
fmul.d f6,f10,f8
```



  - Name dependence with f6

Register Renaming

- Example 3:

```
fdiv.d f0,f2,f4
fadd.d S,f0,f8
fsd    S,0(x1)
fsub.d T,f10,f14
fmul.d f6,f10,T
```

- Now only RAW hazards remain, which can be strictly ordered

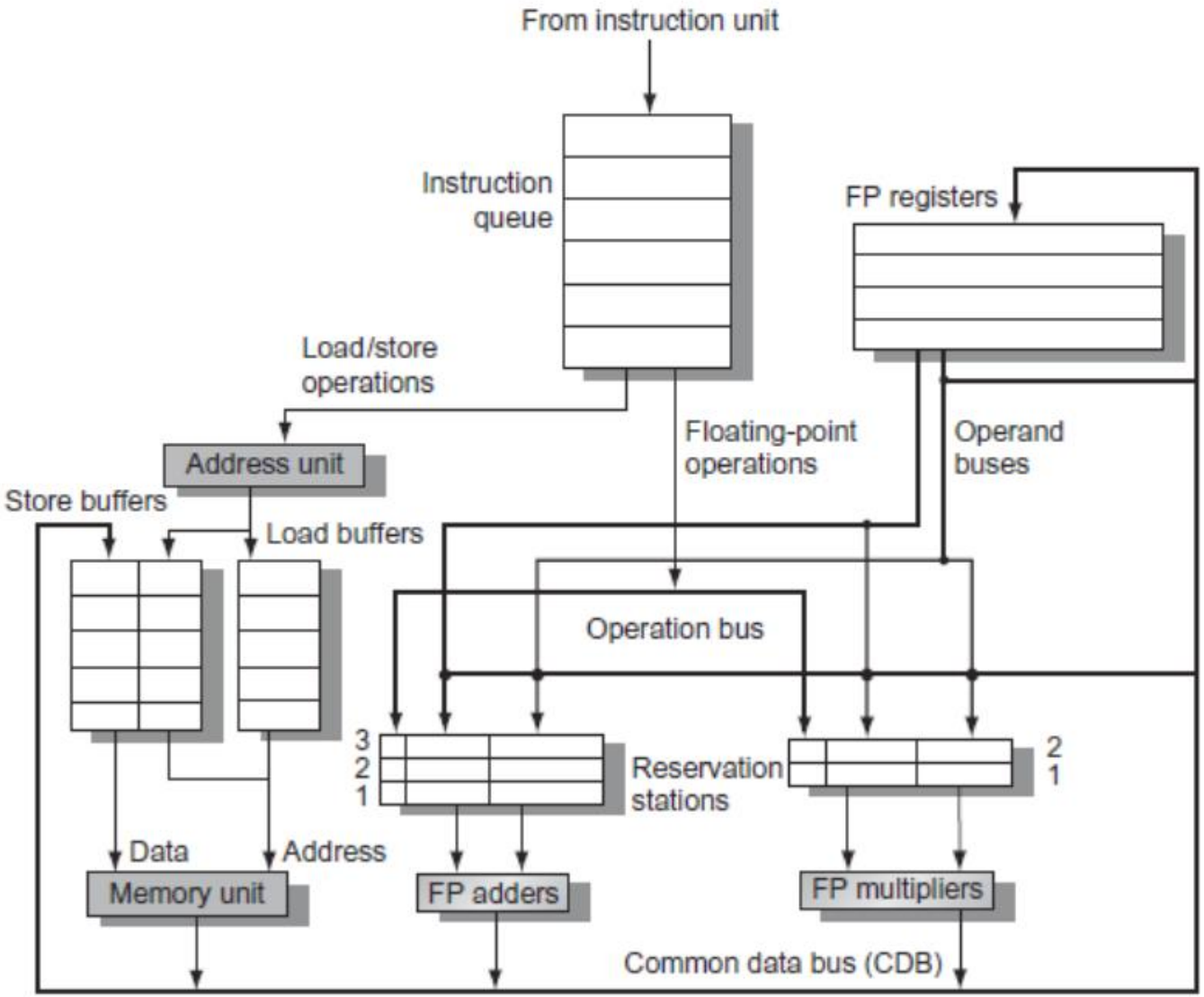
Tomasulo's Algorithm

- Tracks when operands are available
- Introduces register renaming in hardware
  - Minimizes WAW and WAR hazards
- Register renaming is provided by reservation stations (RS)
  - Contains:
    - The instruction
    - Buffered operand values (when available)
    - Reservation station number where the instruction providing the operand values is located

Tomasulo's Algorithm

- RS fetches and buffers an operand as soon as it becomes available (not necessarily involving register file)
- Pending instructions designate the RS that will provide their input
  - Result values broadcast on a result bus, called the common data bus (CDB)
- Only the last output updates the register file
- As instructions are issued, the register specifiers are renamed with the reservation station
- May be more reservation stations than registers
- Load and store buffers
  - Contain data and addresses, act like reservation stations

Tomasulo's Algorithm



Tomasulo's Algorithm Stages

- Fetch (F)
  - Get instructions from the instruction memory to the instruction queue.
- Issue (I)
  - Get next instruction from FIFO queue.
  - If available RS, issue the instruction to the RS with operand values if available from the register file (RF).
  - If operand values not available, stall the instruction.
- Execute (E)
  - When operand becomes available, store it in any reservation stations waiting for it.
  - When all operands are ready, issue the instruction.
  - Loads and store maintained in program order through effective address.
  - No instruction allowed to initiate execution until all branches that proceed it in program order have completed.
- Write result
  - Write result on CDB into RSs, RF, and store buffers
    - (Stores must wait until address and value are received)

Contents

Contents

- Dynamic Scheduling: Examples and Algorithms
  - Example of Floating-Point Operations
  - Loop Example

Example of FP Operations

```
fld     f6,32(x2)
fld     f2,44(x3)
fmul.d  f0,f2,f4
fsub.d  f8,f2,f6
fdiv.d  f0,f0,f6
fadd.d  f6,f8,f2
```

- Latencies:
  - Load 1 cycle (A) Address Calculation and 1 cycle (M) Memory Access
  - Add 2 cycles
  - Mul 6 cycles
  - Div 12 cycles

Information Tables Status

		Instruction status		
Instruction		Issue	Execute	Write result
fld	f6,32(x2)	√	√	√
fld	f2,44(x3)	√	√	
fmul.d	f0,f2,f4	√		
fsub.d	f8,f2,f6	√		
fdiv.d	f0,f0,f6	√		
fadd.d	f6,f8,f2	√		

Status when all instructions have issued but only I1 has written.

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					44 + Regs[x3]
Add1	Yes	SUB		Mem[32 + Regs[x2]]	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		Regs[f4]	Load2		
Mult2	Yes	DIV		Mem[32 + Regs[x2]]	Mult1		

	Register status								
Field	f0	f2	f4	f6	f8	f10	f12	...	f30
Qi	Mult1	Load2		Add2	Add1	Mult2			

Loop Example

```
Loop: fld    f0,0(x1)
      fmul.d f4,f0,f2
      fsd    f4,0(x1)
      addi   x1,x1,-8
      bne    x1,x2,Loop
         // branches if x1 ≠ x2
```

Information Tables Status

		Instruction status			
Instruction		From iteration	Issue	Execute	Write result
fld	f0,0(x1)	1	√	√	
fmul.d	f4,f0,f2	1	√		
fsd	f4,0(x1)	1	√		
fld	f0,0(x1)	2	√	√	
fmul.d	f4,f0,f2	2	√		
fsd	f4,0(x1)	2	√		

Status when all instrs. have issued but only the loads have executed.

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	Yes	Load					Regs[x1] + 0
Load2	Yes	Load					Regs[x1] – 8
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL		Regs[f2]	Load1		
Mult2	Yes	MUL		Regs[f2]	Load2		
Store1	Yes	Store	Regs[x1]			Mult1	
Store2	Yes	Store	Regs[x1] – 8			Mult2	

	Register status								
Field	f0	f2	f4	f6	f8	f10	f12	...	f30
Qi	Load2		Mult2						

Contents

Contents

- Hardware-Based Speculation
  - Introduction
  - Reorder Buffer (ROB)
  - Stages with Reorder Buffer
  - Commit Stage
  - Examples

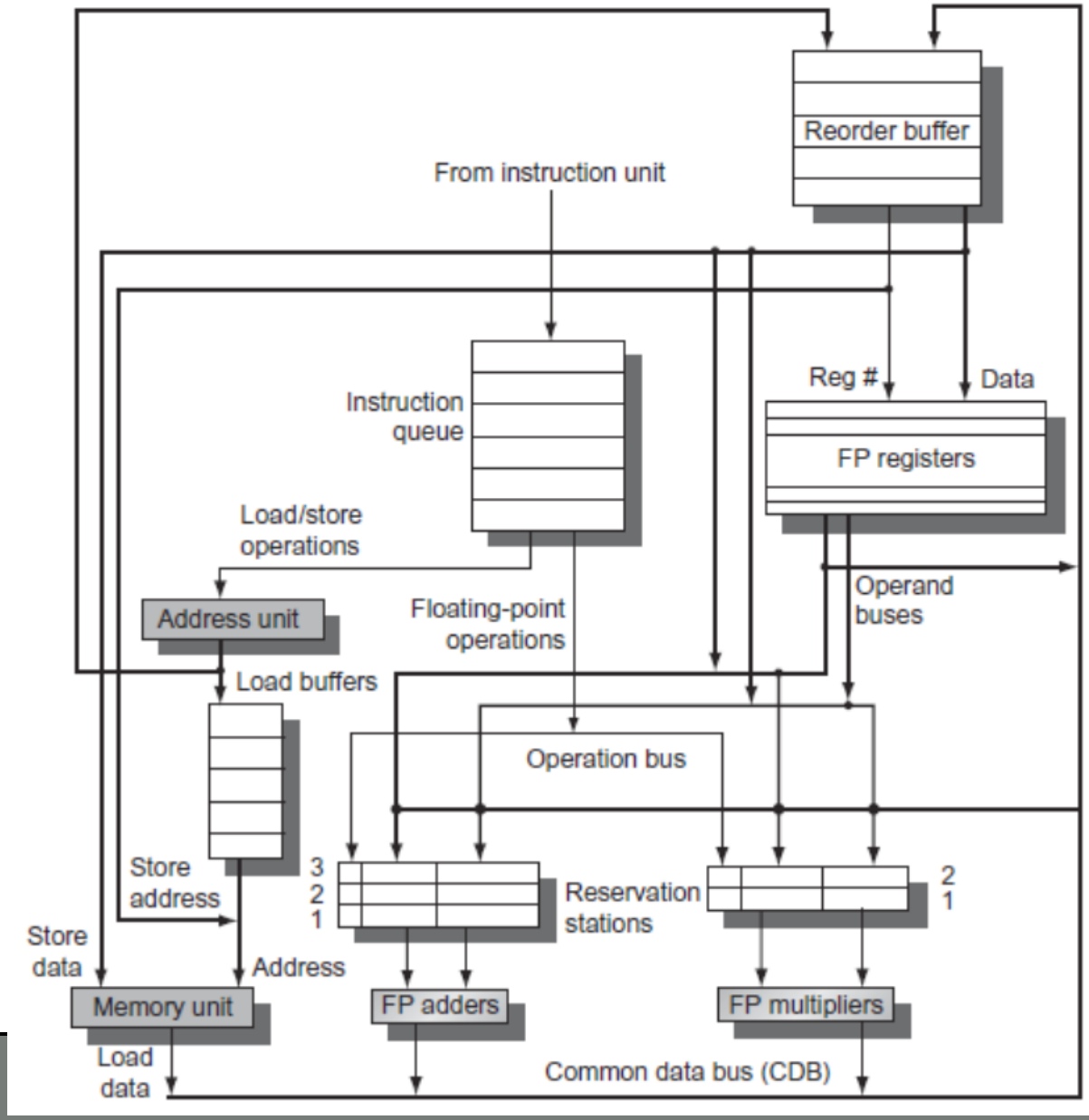
Introduction

- Execute instructions along predicted execution paths but only commit the results if prediction was correct.
- Instruction commit: allowing an instruction to update the register file when instruction is no longer speculative.
- Need an additional piece of hardware to prevent any irrevocable action until an instruction commits.
  - I.e. updating state

Reorder Buffer (ROB)

- Reorder buffer – holds the result of instruction between completion and commit
- Four fields:
  - Instruction type: branch/store/register
  - Destination field: register number
  - Value field: output value
  - Ready field: completed execution?
- Modify reservation stations:
  - Operand source is now reorder buffer instead of reservation stations.

Reorder Buffer



Stages with Reorder Buffer

- Fetch (F)
- Issue (I):
  - Allocate RS and ROB, read available operands
- Execute (E):
  - Begin execution when operand values are available
- Write result (W):
  - Write result and ROB tag on CDB
- Commit (C):
  - When ROB reaches head of ROB, update RF.
  - When a mispredicted branch reaches head of ROB, discard all entries.

Commit Stage

- Register values and memory values are not written until an instruction commits
- On misprediction:
  - Speculated entries in ROB are cleared
- Exceptions:
  - Not recognized until it is ready to commit

Example of FP Operations

```
fld     f6,32(x2)
fld     f2,44(x3)
fmul.d  f0,f2,f4
fsub.d  f8,f2,f6
fdiv.d  f0,f0,f6
fadd.d  f6,f8,f2
```

- Latencies:
  - Load 1 cycle (A) Address Calculation and 1 cycle (M) Memory Access
  - Add 2 cycles
  - Mul 6 cycles
  - Div 12 cycles

Information Tables Status

Reorder buffer

Entry	Busy	Instruction		State	Destination	Value
1	No	fld	f6,32(x2)	Commit	f6	Mem[32 + Regs[x2]]
2	No	fld	f2,44(x3)	Commit	f2	Mem[44 + Regs[x3]]
3	Yes	fmul.d	f0,f2,f4	Write result	f0	#2 × Regs[f4]
4	Yes	fsub.d	f8,f2,f6	Write result	f8	#2 − #1
5	Yes	fdiv.d	f0,f0,f6	Execute	f0	
6	Yes	fadd.d	f6,f8,f2	Write result	f6	#4 + #2

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	No							
Load2	No							
Add1	No							
Add2	No							
Add3	No							
Mult1	No	fmul.d	Mem[44 + Regs[x3]]	Regs[f4]			#3	
Mult2	Yes	fdiv.d		Mem[32 + Regs[x2]]	#3		#5	

Status when all the multiply instruction is ready to commit.

FP register status

Field	f0	f1	f2	f3	f4	f5	f6	f7	f8	f10
Reorder #	3						6		4	5
Busy	Yes	No	No	No	No	No	Yes	…	Yes	Yes

Assumes fdiv.d writes to f10

Loop Example

Using pipeline diagrams, show the execution of two iterations of this loop assuming the branch is predicted taken and is resolved as not taken.

```
Loop: fld     f0,0(x1)
      fmul.d  f4,f0,f2
      fsd     f4,0(x1)
      addi    x1,x1,-8
      bne     x1,x2,Loop
         // branches if x1 ≠ x2
```

Contents

Contents

- Exploiting ILP Using Multiple Issue and Static Scheduling
  - Introduction
  - Multiple Issue Approaches
  - VLIW Processors

Introduction

- To achieve CPI < 1, need to complete multiple instructions per clock

- Solutions:
  - Statically scheduled superscalar processors
  - Dynamically scheduled superscalar processors
  - VLIW (very long instruction word) processors

Multiple Issue Approaches

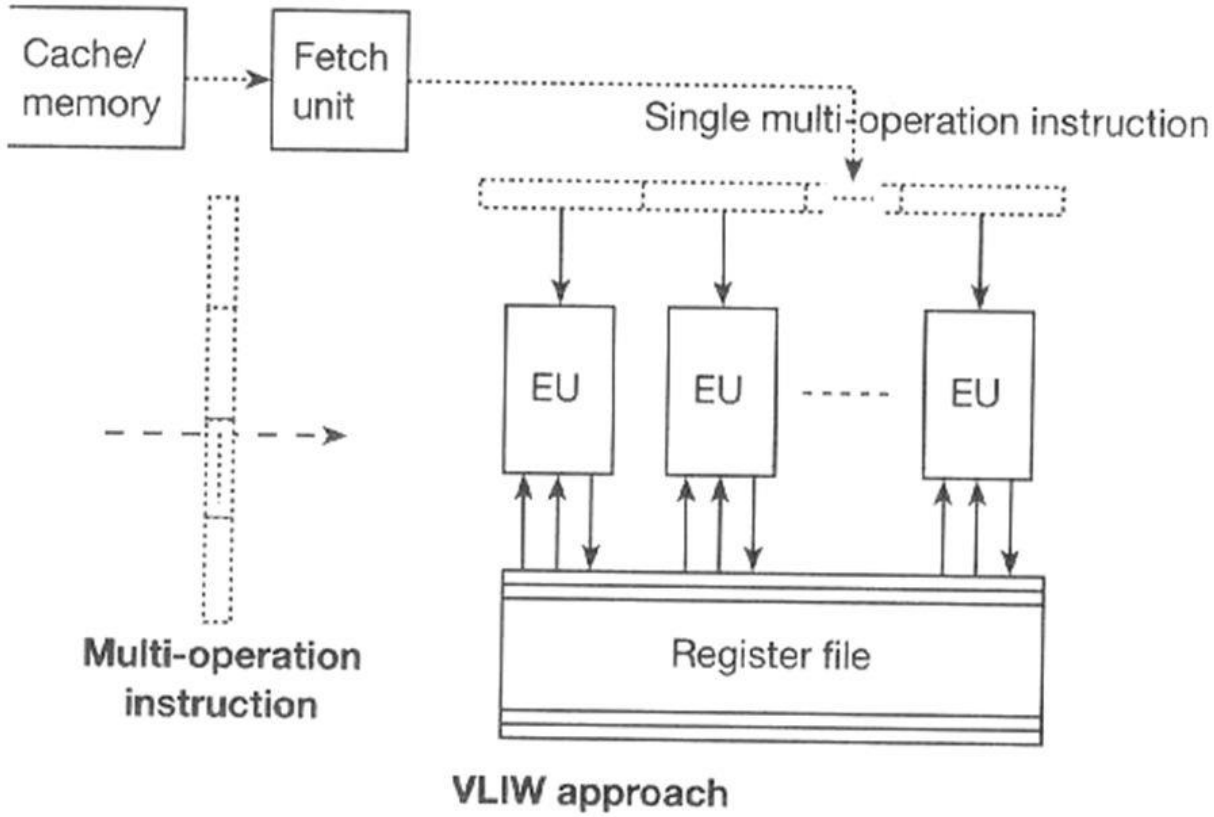
Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, 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

VLIW Processors

- Package multiple operations into one instruction
- Example VLIW processor:
  - One integer instruction (or branch)
  - Two independent floating-point operations
  - Two independent memory references
- Must be enough parallelism in code to fill the available slots

VILW Processors

VLIW
(very long instruction word,1024 bits!)



VLIW approach

VLIW Processors

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
fld f0,0(x1)	fld f6,-8(x1)			
fld f10,-16(x1)	fld f14,-24(x1)			
fld f18,-32(x1)	fld f22,-40(x1)	fadd.d f4,f0,f2	fadd.d f8,f6,f2	
fld f26,-48(x1)		fadd.d f12,f0,f2	fadd.d f16,f14,f2	
		fadd.d f20,f18,f2	fadd.d f24,f22,f2	
fsd f4,0(x1)	fsd f8,-8(x1)	fadd.d f28,f26,f24		
fsd f12,-16(x1)	fsd f16,-24(x1)			addi x1,x1,-56
fsd f20,24(x1)	fsd f24,16(x1)			
fsd f28,8(x1)				bne x1,x2,Loop

- Disadvantages:
  - Statically finding parallelism
  - Code size
  - No hazard detection hardware
  - Binary code compatibility

Contents

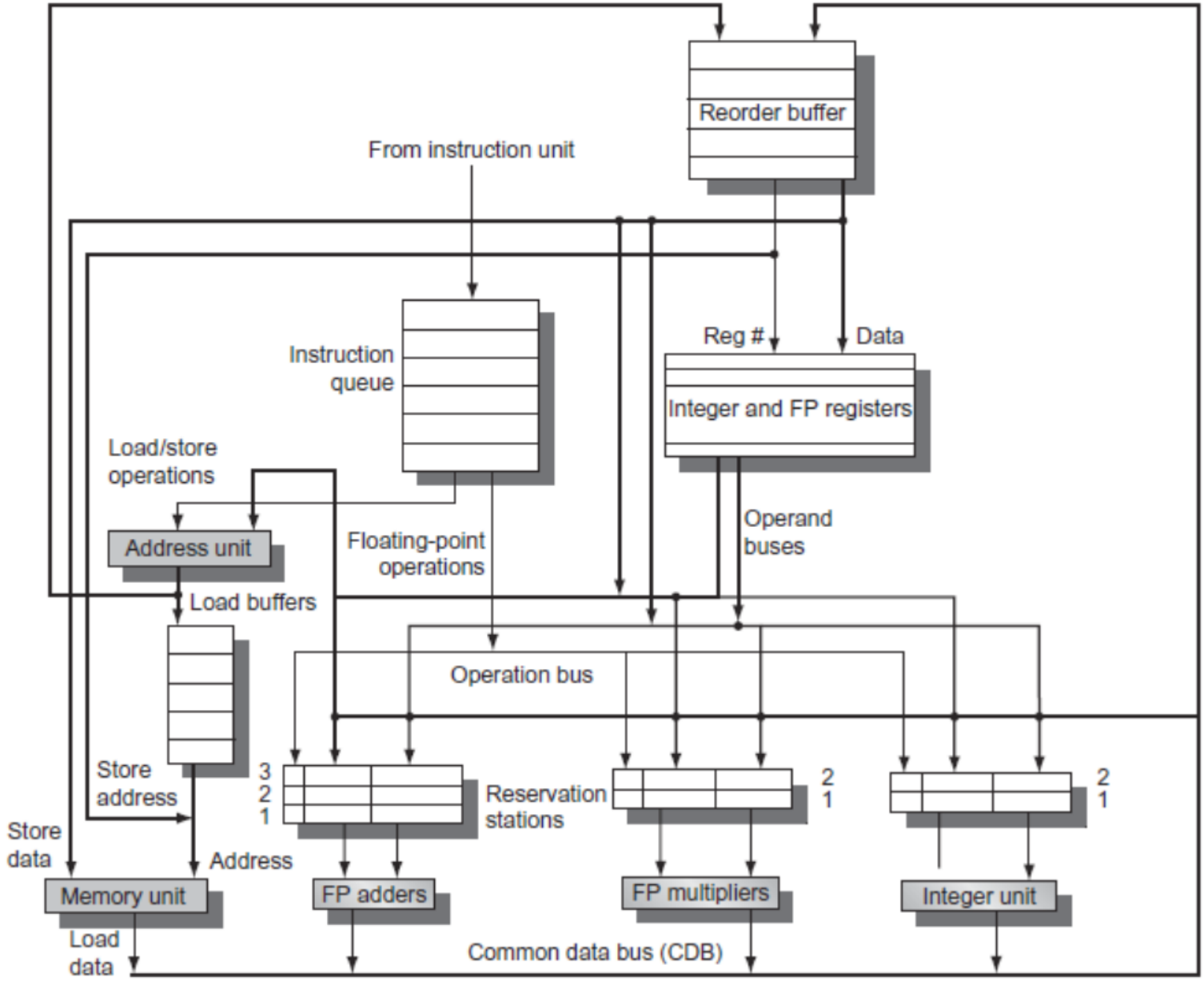
Contents

- Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation
  - Introduction
  - Multiple Issue with Speculation
  - Example

Introduction

- Modern microarchitectures:
  - Dynamic scheduling + multiple issue + speculation
- Two approaches:
  - Assign reservation stations and update pipeline control table in half clock cycles
    - Only supports 2 instructions/clock
  - Design logic to handle any possible dependencies between the instructions
- Issue logic is the bottleneck in dynamically scheduled superscalars

Multiple Issue with Speculation



Multiple Issue with Speculation

- Examine all the dependencies among the instructions in the bundle
- If dependencies exist in bundle, encode them in reservation stations
- Also need multiple completion/commit
- To simplify RS allocation:
  - Limit the number of instructions of a given class that can be issued in a “bundle”, i.e. one FP, one integer, one load, one store

Example

Using pipeline diagrams, show the execution of three iterations of this loop assuming superscaler degree of 2.

```
Loop:
     ld   x2,0(x1)    //x2=array element
     addi x2,x2,1     //increment x2
     sd   x2,0(x1)    //store result
     addi x1,x1,8     //increment pointer
     bne  x2,x3,Loop  //branch if not last
```

Example

Iteration number	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	ld x2,0(x1)	1	2	3	4	5	First issue
1	addi x2,x2,1	1	5		6	7	Wait for ld
1	sd x2,0(x1)	2	3			7	Wait for addi
1	addi x1,x1,8	2	3		4	8	Commit in order
1	bne x2,x3,Loop	3	7			8	Wait for addi
2	ld x2,0(x1)	4	5	6	7	9	No execute delay
2	addi x2,x2,1	4	8		9	10	Wait for ld
2	sd x2,0(x1)	5	6			10	Wait for addi
2	addi x1,x1,8	5	6		7	11	Commit in order
2	bne x2,x3,Loop	6	10			11	Wait for addi
3	ld x2,0(x1)	7	8	9	10	12	Earliest possible
3	addi x2,x2,1	7	11		12	13	Wait for ld
3	sd x2,0(x1)	8	9			13	Wait for addi
3	addi x1,x1,8	8	9		10	14	Executes earlier
3	bne x2,x3,Loop	9	13			14	Wait for addi

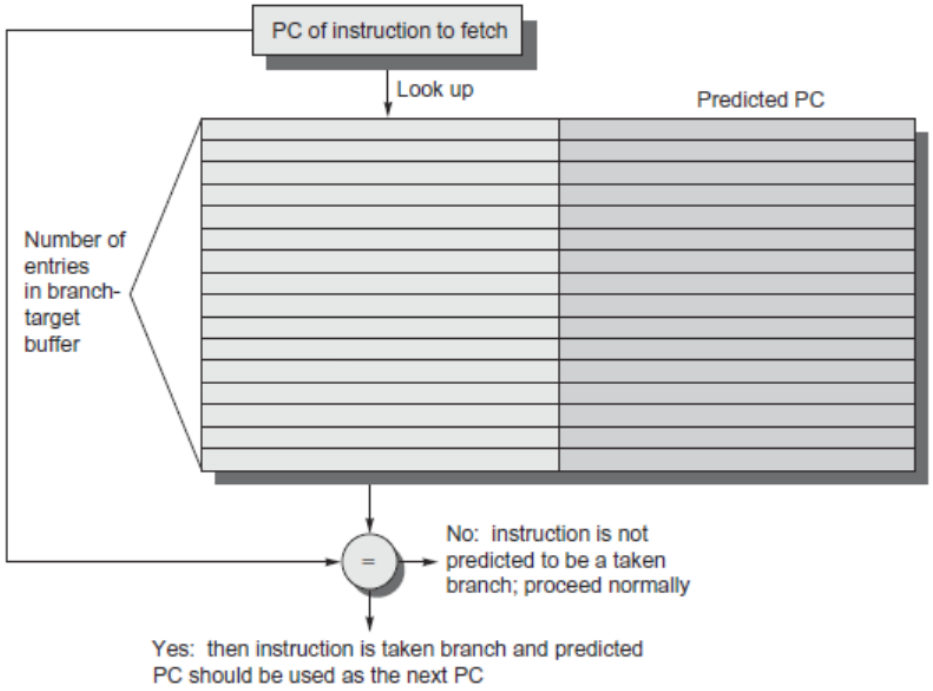
Contents

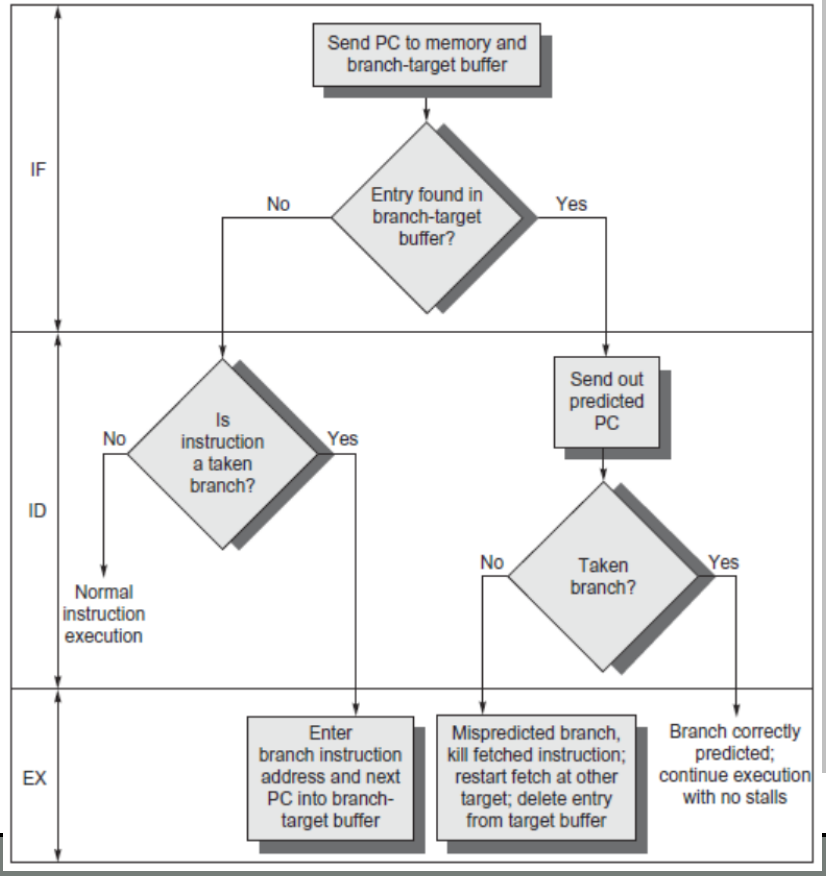
Contents

- Advanced Techniques for Instruction Delivery and Speculation
  - Branch-Target Buffer
  - Branch Folding
  - Return Address Predictor
  - Integrated Instruction Fetch Units
  - Hardware Register Renaming
  - How Much Speculation?
  - Value and Address Prediction

Branch-Target Buffer

- Need high instruction bandwidth
  - Branch-Target Buffers (BTB)
    - Next PC prediction buffer, indexed by current PC





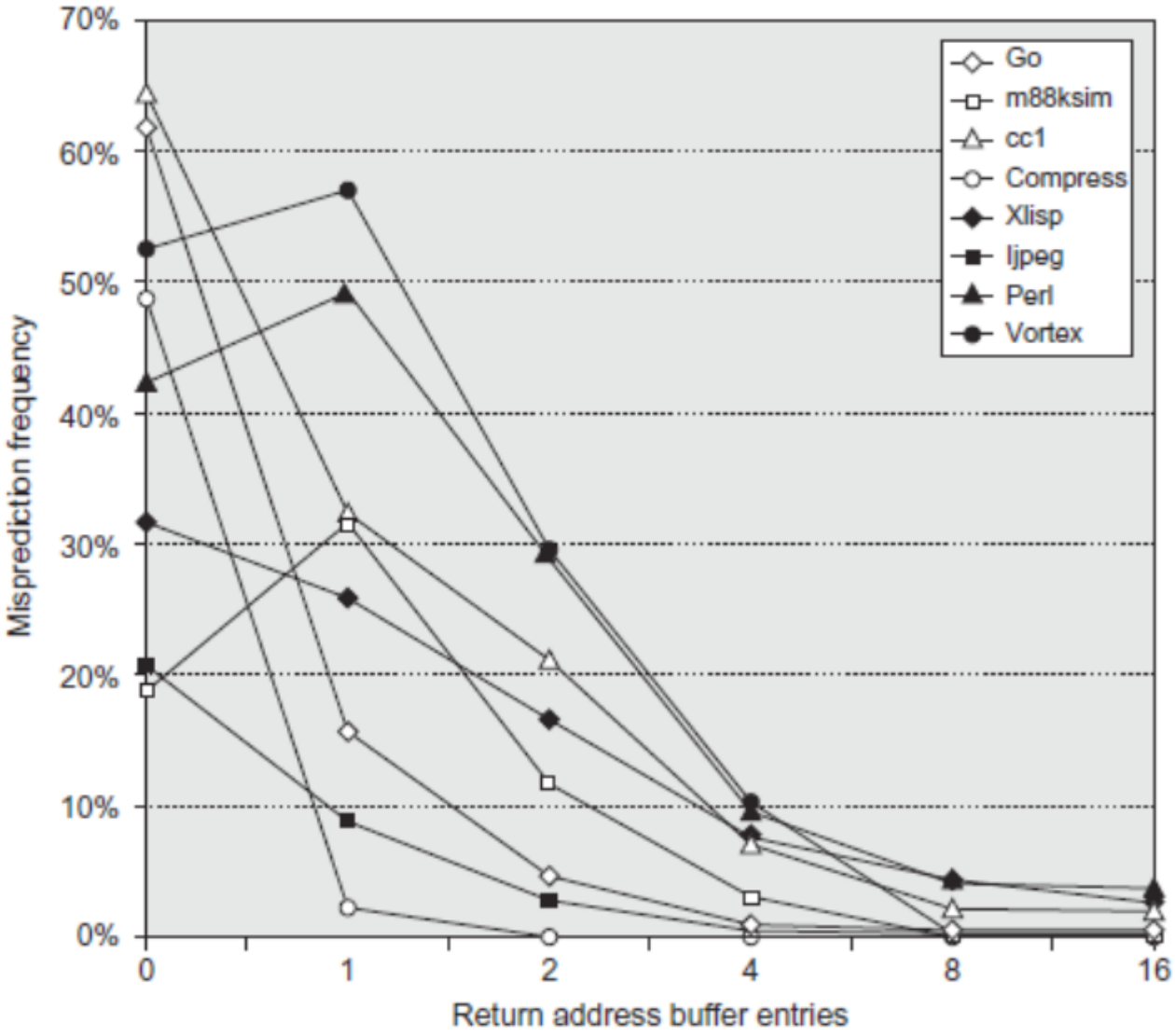
Branch Folding

- Optimization:
  - Larger branch-target buffer
  - Add target instruction into buffer to deal with longer decoding time required by larger buffer
  - “Branch folding”

Return Address Predictor

- Most unconditional branches come from function returns
- The same procedure can be called from multiple sites
  - Causes the buffer to potentially forget about the return address from previous calls
- Create return address buffer organized as a stack

Return Address Predictor



Integrated Instruction Fetch Unit

- Design monolithic unit that performs:
  - Branch prediction
  - Instruction prefetch
    - Fetch ahead
  - Instruction memory access and buffering
    - Deal with crossing cache lines

Hardware Register Renaming

- Instead of virtual registers from reservation stations and reorder buffer, create a single register pool
  - Contains visible registers and virtual registers
- Use hardware-based map to rename registers during issue
- WAW and WAR hazards are avoided
- Speculation recovery occurs by copying during commit
- Still need a ROB-like queue to update table in order
- Simplifies commit:
  - Record that mapping between architectural register and physical register is no longer speculative
  - Free up physical register used to hold older value
  - In other words: SWAP physical registers on commit
- Physical register de-allocation is more difficult
  - Simple approach: deallocate virtual register when next instruction writes to its mapped architecturally-visibly register

Hardware Register Renaming

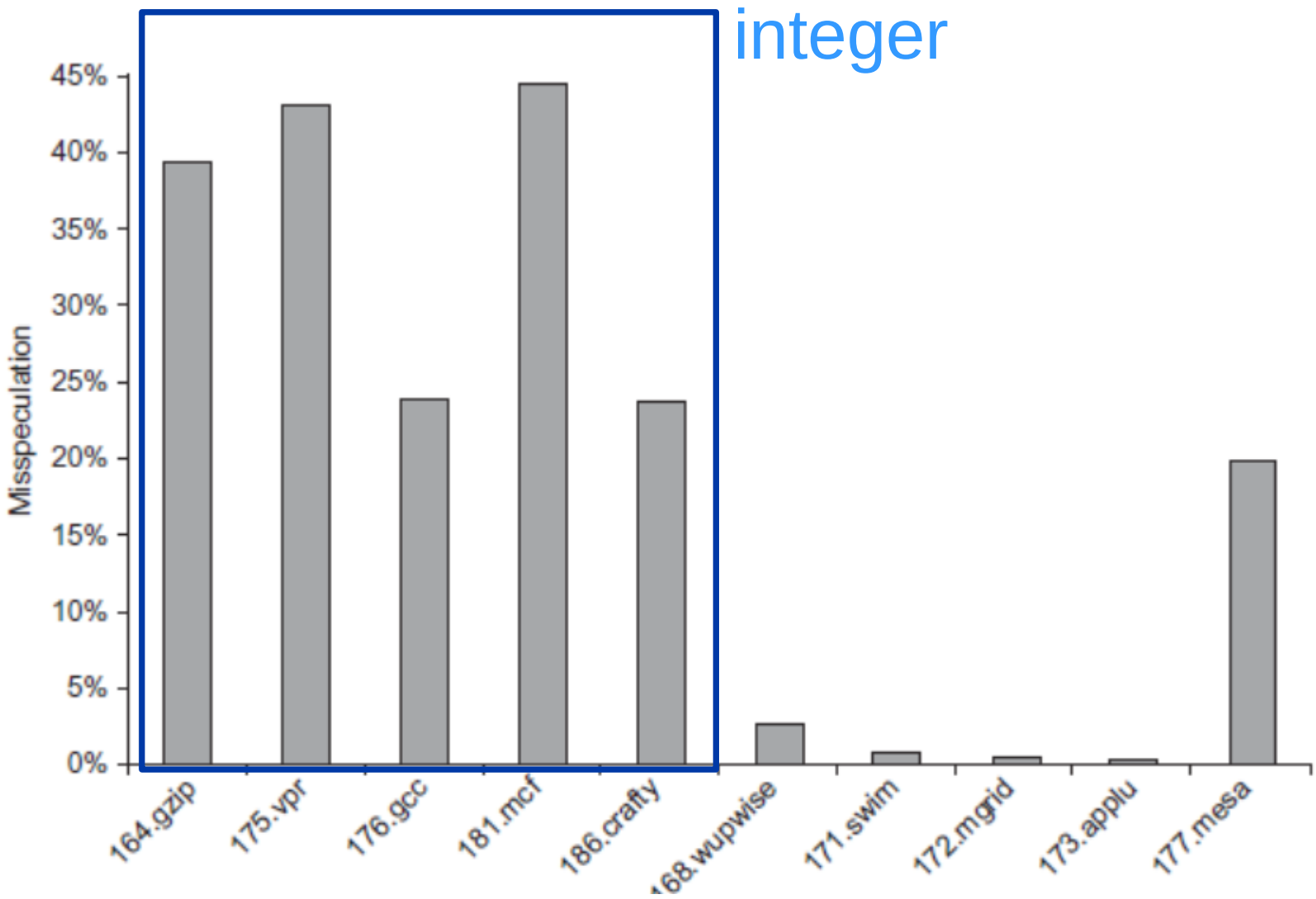
- Combining instruction issue with register renaming:
  - Issue logic pre-reserves enough physical registers for the bundle
  - Issue logic finds dependencies within bundle, maps registers as necessary
  - Issue logic finds dependencies between current bundle and already in-flight bundles, maps registers as necessary

Instr. #	Instruction	Physical register assigned or destination	Instruction with physical register numbers	Rename map changes
1	add x1,x2,x3	p32	add p32,p2,p3	x1-> p32
2	sub x1,x1,x2	p33	sub p33,p32,p2	x1->p33
3	add x2,x1,x2	p34	add p34,p33,x2	x2->p34
4	sub x1,x3,x2	p35	sub p35,p3,p34	x1->p35
5	add x1,x1,x2	p36	add p36,p35,p34	x1->p36
6	sub x1,x3,x1	p37	sub p37,p3,p36	x1->p37

How Much Speculation?

- How much to speculate
  - Mis-speculation degrades performance and power relative to no speculation
    - May cause additional misses (cache, TLB)
  - Prevent speculative code from causing higher costing misses (e.g. L2)
- Speculating through multiple branches
  - Complicates speculation recovery
- Speculation and energy efficiency
  - Note: speculation is only energy efficient when it significantly improves performance

How Much Speculation?

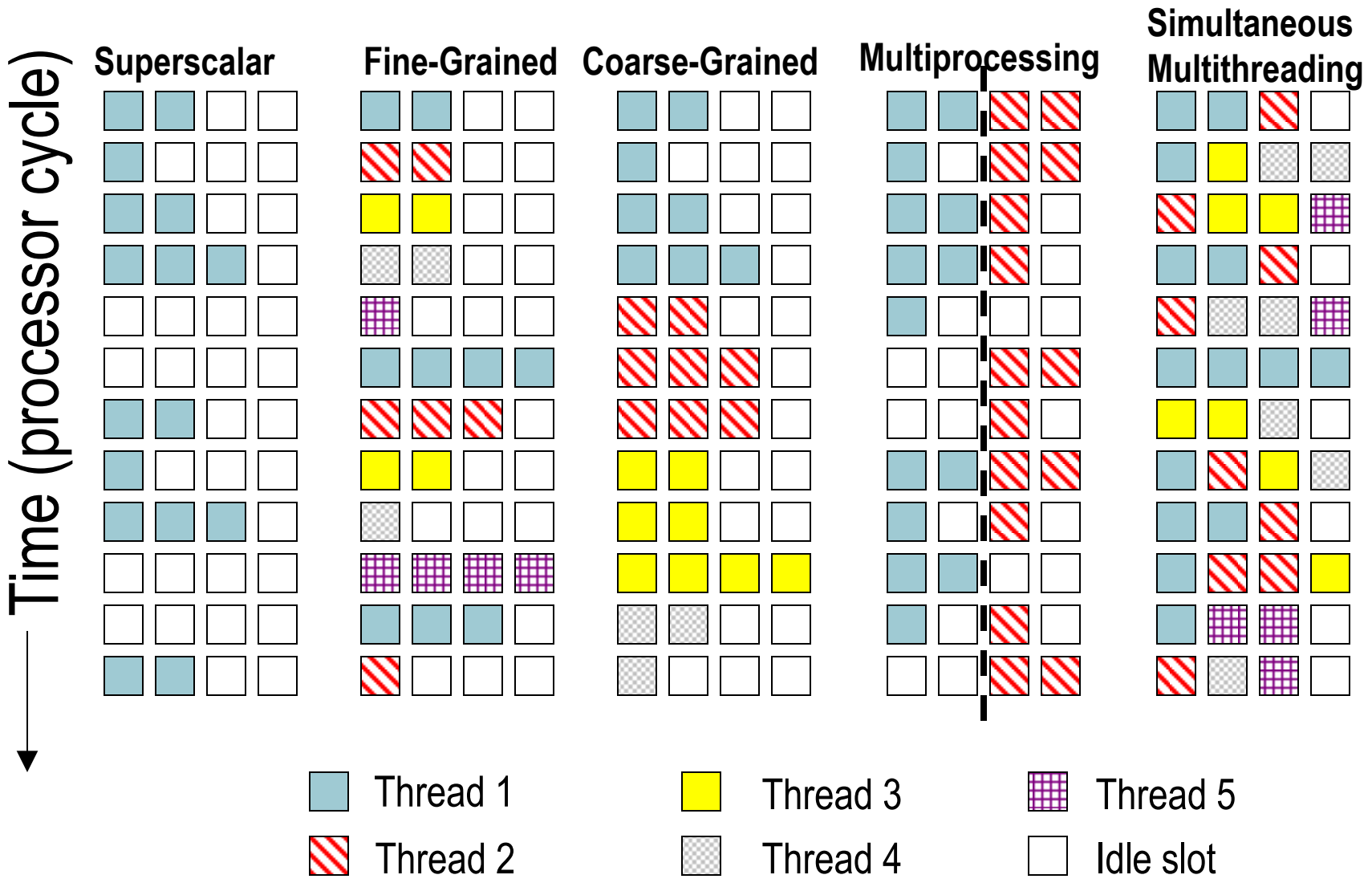


Value and Address Prediction

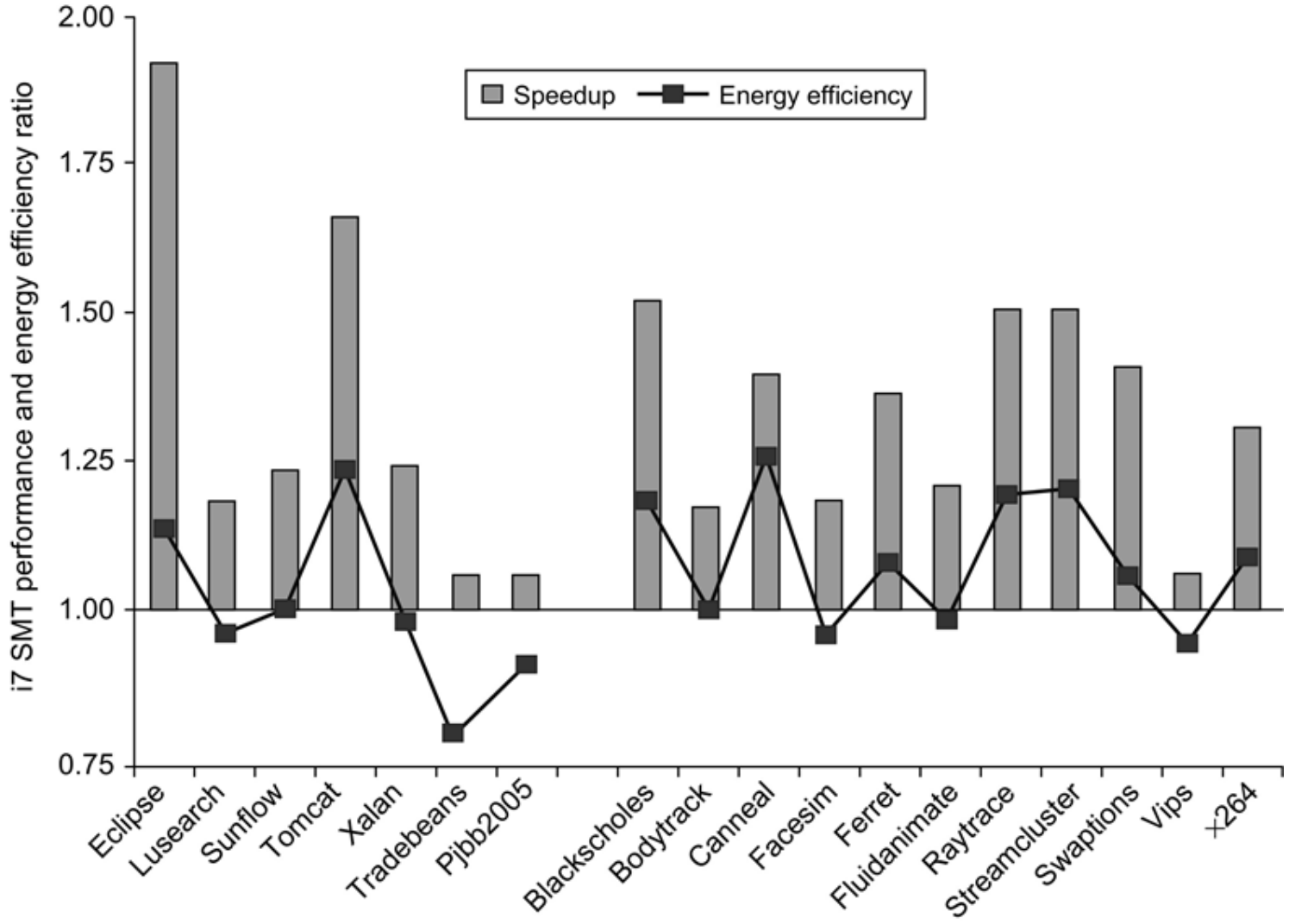
- Value prediction
  - Uses:
    - Loads that load from a constant pool
    - Instruction that produces a value from a small set of values
  - Not incorporated into modern processors
- Similar idea--*address aliasing prediction*--is used on some processors to determine if two stores or a load and a store reference the same address to allow for reordering

Contents

Multithreaded Approaches



Intel Core i7 SMT Evaluation



Contents

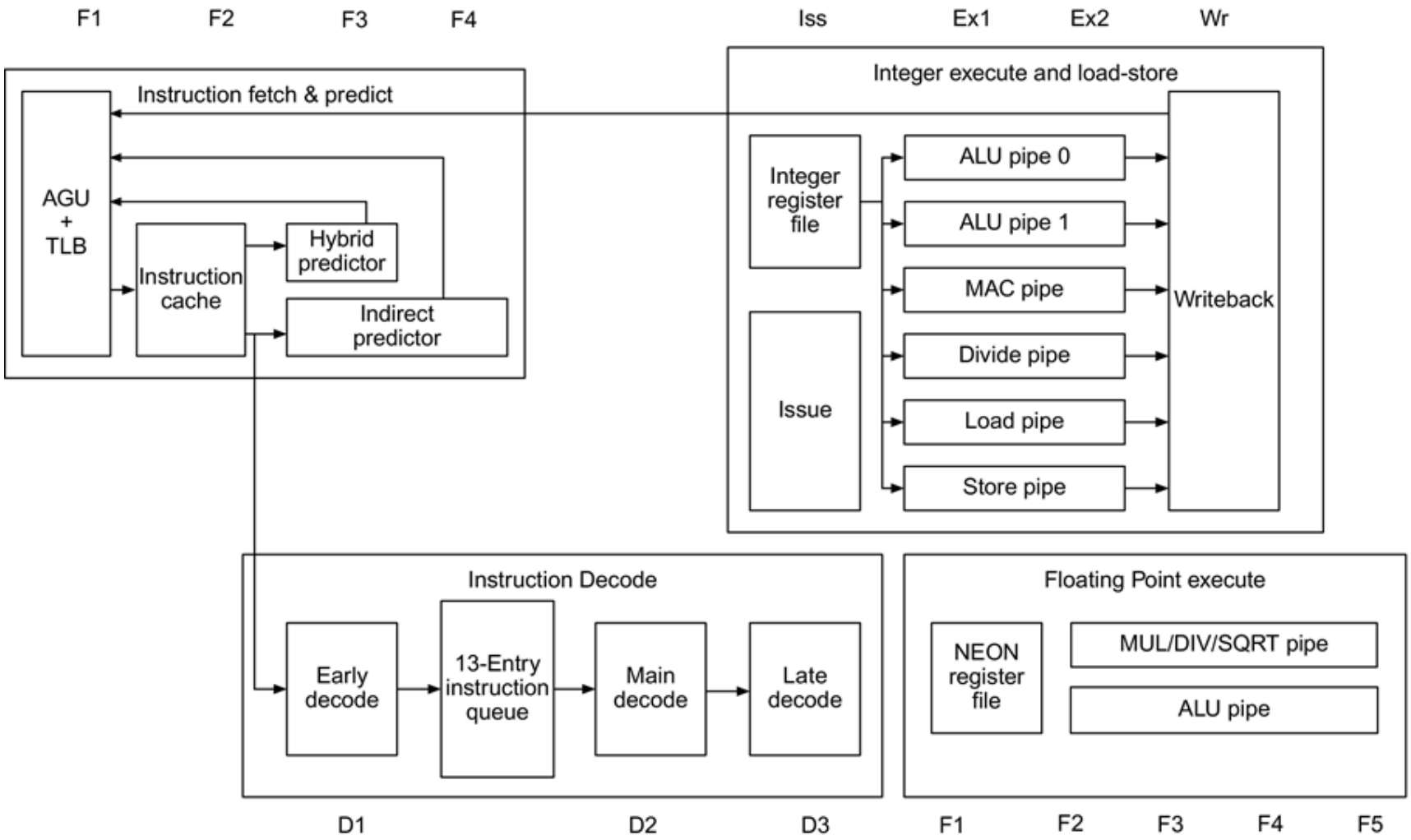
Contents

- The Intel i7 6700 and ARM Cortex-A53
  - ARM Cortex-A53
    - Overview
    - Branch Prediction
    - Performance
  - Intel Core i7 6700
    - Overview
    - Core i7 920 vs 6700

ARM Cortex-A53 Overview

- Dual-issue, statically scheduled superscalar with dynamic issue detection.
- 8 stages for integer instructions (F1, F2, D1, D2, D3/ISS, EX1, EX2, WB).
- 10 stages for FP instructions (F1, F2, D1, D2, D3, F1, F2, F3, F4, F5).
- Pipeline stalls due to:
  - Functional hazards: two instrs. same unit
  - Data hazards: two dependent instrs
  - Control hazards: branch misprediction

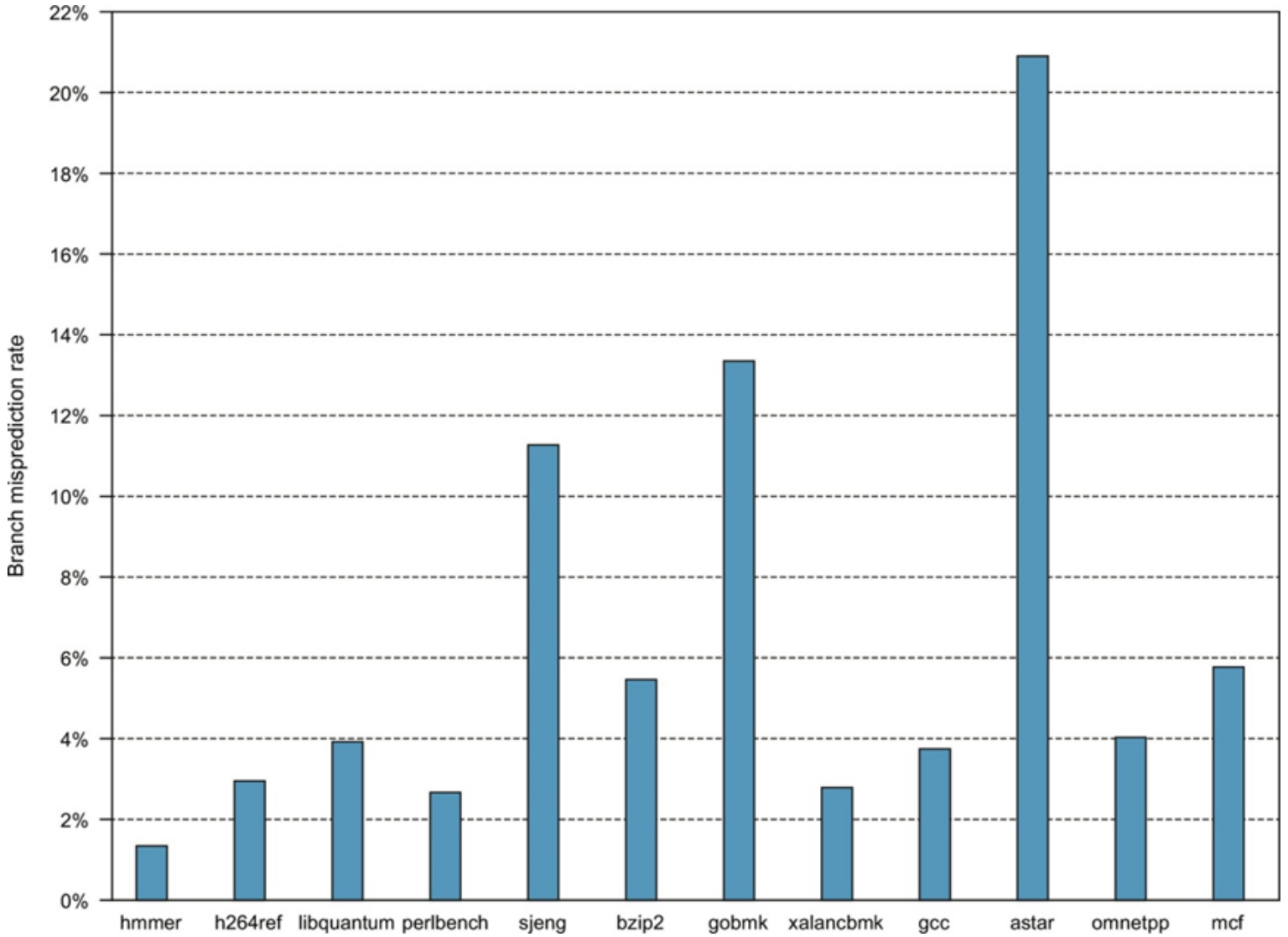
ARM Cortex-A53 Overview



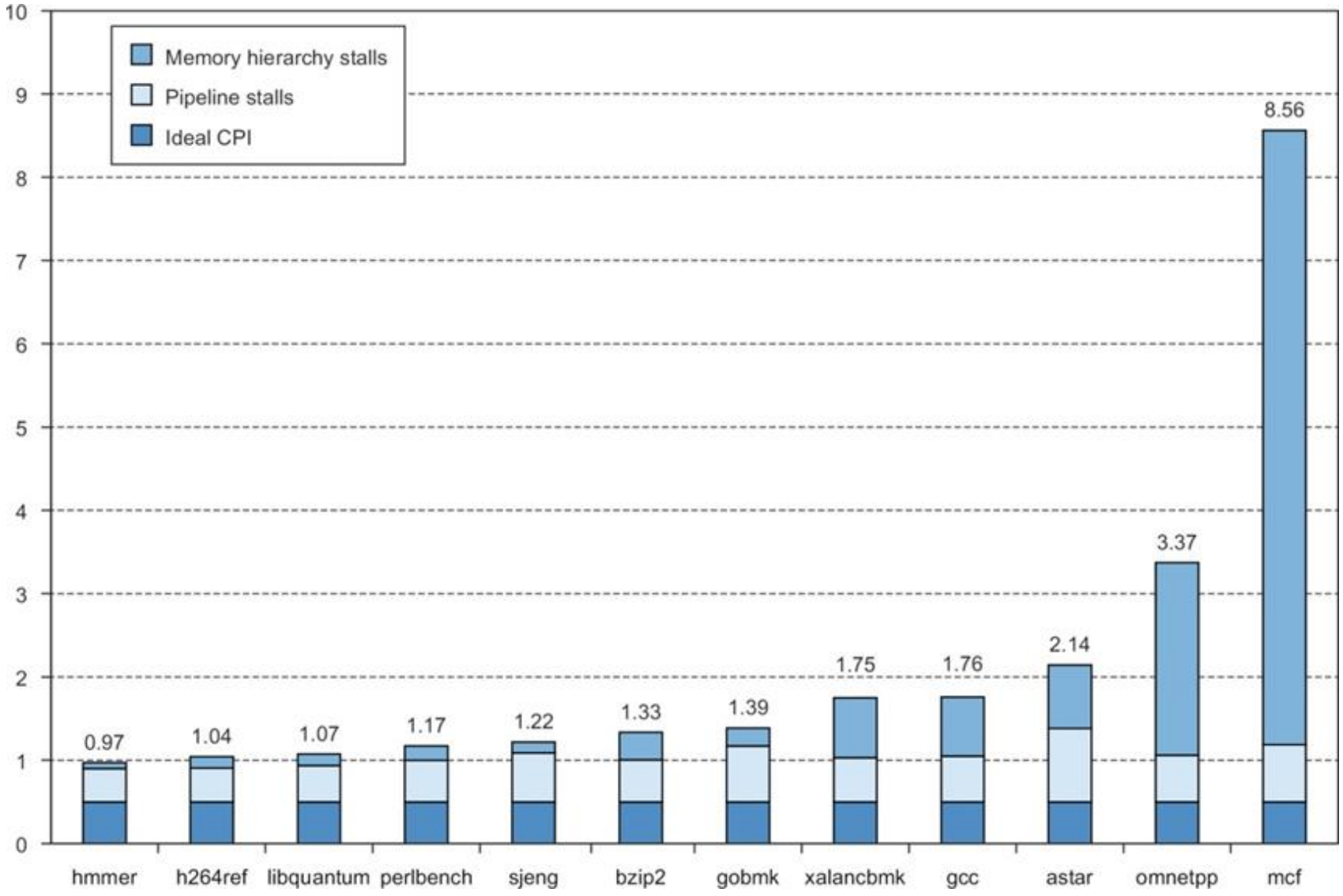
A53 Branch Prediction

- The instruction fetch unit has
  - Single entry Branch Target Instruction Cache (gives the next two target instrs), no delay
  - Hybrid Predictor: Global predictor with branch history registers and a 3072-entry branch history table, 2-cycle delay
  - Indirect Predictor: 256-entry Branch Target Address Cache, 3-cycle delay.
  - Return Address stack: 8 entries, 3-cycle delay.
- Branch decisions are made in ALU Pipe 0.
- Misprediction penalty is 8 cycles.

A53 Misprediction, SPECint2006



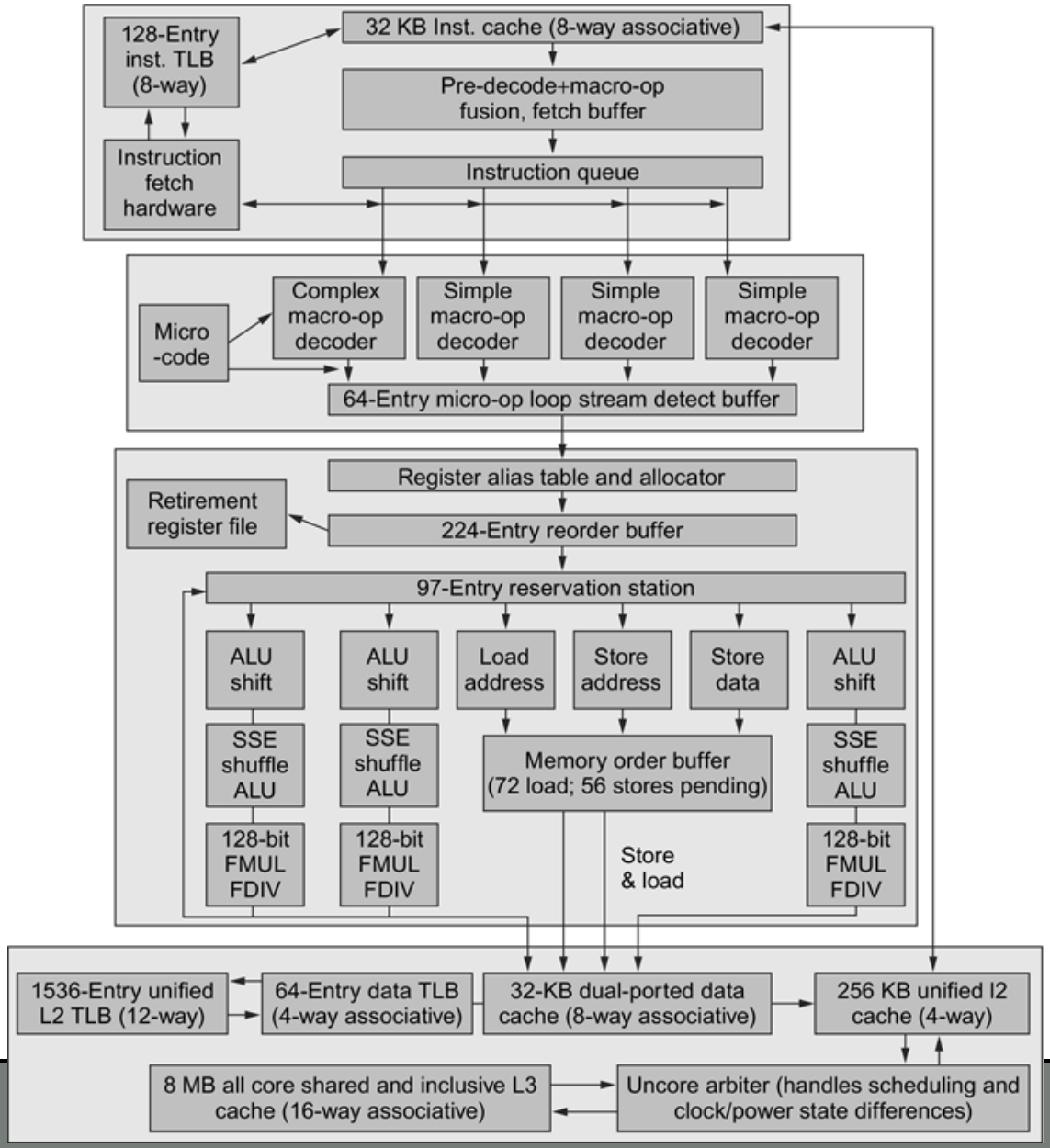
A53 Performance (CPI)



Intel Core i7 6700 Overview

- Out-of-order speculative superscalar, degree 4.
- 14 stages
  - Instruction Fetch: Fetches 16 bytes, has branch prediction and return address stack.
  - 16-entry instruction buffer for predecoded, fused macro-ops
  - 4 macro-ops decoders (one for complex instrs).
  - 64-entry micro-op buffer with loop stream detection and fusion.
  - Instruction issue with register renaming to 224-ROB and 97-RS.
  - 6 functional units

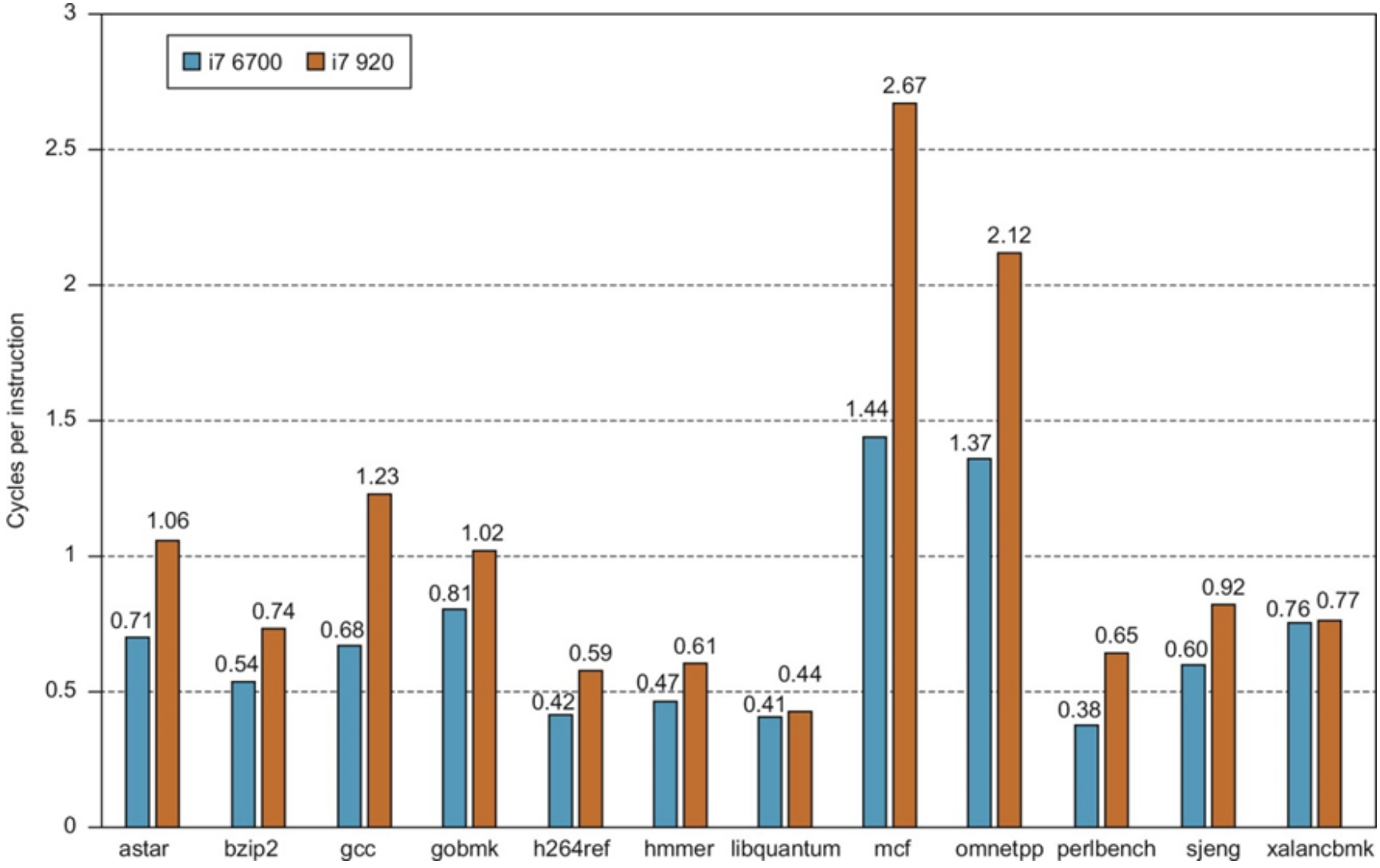
Intel Core i7 6700



i7 920 (1st Gen.) vs 6700 (6th Gen.)

Resource	i7 920 (Nehalem)	i7 6700 (Skylake)
Micro-op queue (per thread)	28	64
Reservation stations	36	97
Integer registers	NA	180
FP registers	NA	168
Outstanding load buffer	48	72
Outstanding store buffer	32	56
Reorder buffer	128	256

i7 920 (1st Gen.) vs 6700 (6th Gen.)



Contents

Fallacies and Pitfalls

- F: Processors with lower CPIs / faster clock rates will also be faster

Processor	Implementation technology	Clock rate	Power	SPECCInt2006 base	SPECCFP2006 baseline
Intel Pentium 4 670	90 nm	3.8 GHz	115 W	11.5	12.2
Intel Itanium 2	90 nm	1.66 GHz	104 W approx. 70 W one core	14.5	17.3
Intel i7 920	45 nm	3.3 GHz	130 W total approx. 80 W one core	35.5	38.4

  - Pentium 4 (20 stages) had higher clock, but worse CPI
  - Itanium had same CPI as i7, lower clock than P4

Fallacies and Pitfalls

- P: Sometimes bigger and dumber is better
  - Pentium 4 and Itanium were advanced designs, but could not achieve their peak instruction throughput because of relatively small caches as compared to i7
- P: And sometimes smarter is better than bigger and dumber
  - TAGE branch predictor outperforms gshare with less stored predictions

Fallacies and Pitfalls

- P: Believing that there are large amounts of ILP available, if only we had the right techniques