CMCS 611-101
Advanced Computer Architecture

Lecture 8

Control Hazards and Exception Handling

September 30, 2009

www.csee.umbc.edu/~younis/CMSC611/CMSC611.htm

Lecture's Overview

- ***Previous Lecture:***
  - ➔ Data Hazards
    - Forwarding techniques for simple data hazards resolution
    - Data hazards classifications and detection logic
    - Load-caused pipeline stalls and how to limit their scope
    - Compiler-based instruction scheduling to avoid pipeline stalls
    - Implementation of data hazard detection and forwarding logic
- ***This Lecture***
  - ➔ Control hazards
  - ➔ Pipelining and exception handling

Pipeline Hazards

- Pipeline hazards are cases that affect instruction execution semantics and thus need to be detected and corrected
- Hazards types

Structural hazard: attempt to use a resource two different ways at same time

- ➔ E.g., combined washer/dryer would be a structural hazard or folder busy doing something else (watching TV)
- ➔ Single memory for instruction and data

Data hazard: attempt to use item before it is ready

- ➔ E.g., one sock of pair in dryer and one in washer; can't fold until get sock from washer through dryer
- ➔ instruction depends on result of prior instruction still in the pipeline

Control hazard: attempt to make a decision before condition is evaluated

- ✓ ➔ E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in
- ➔ branch instructions

- Hazards can always be resolved by waiting

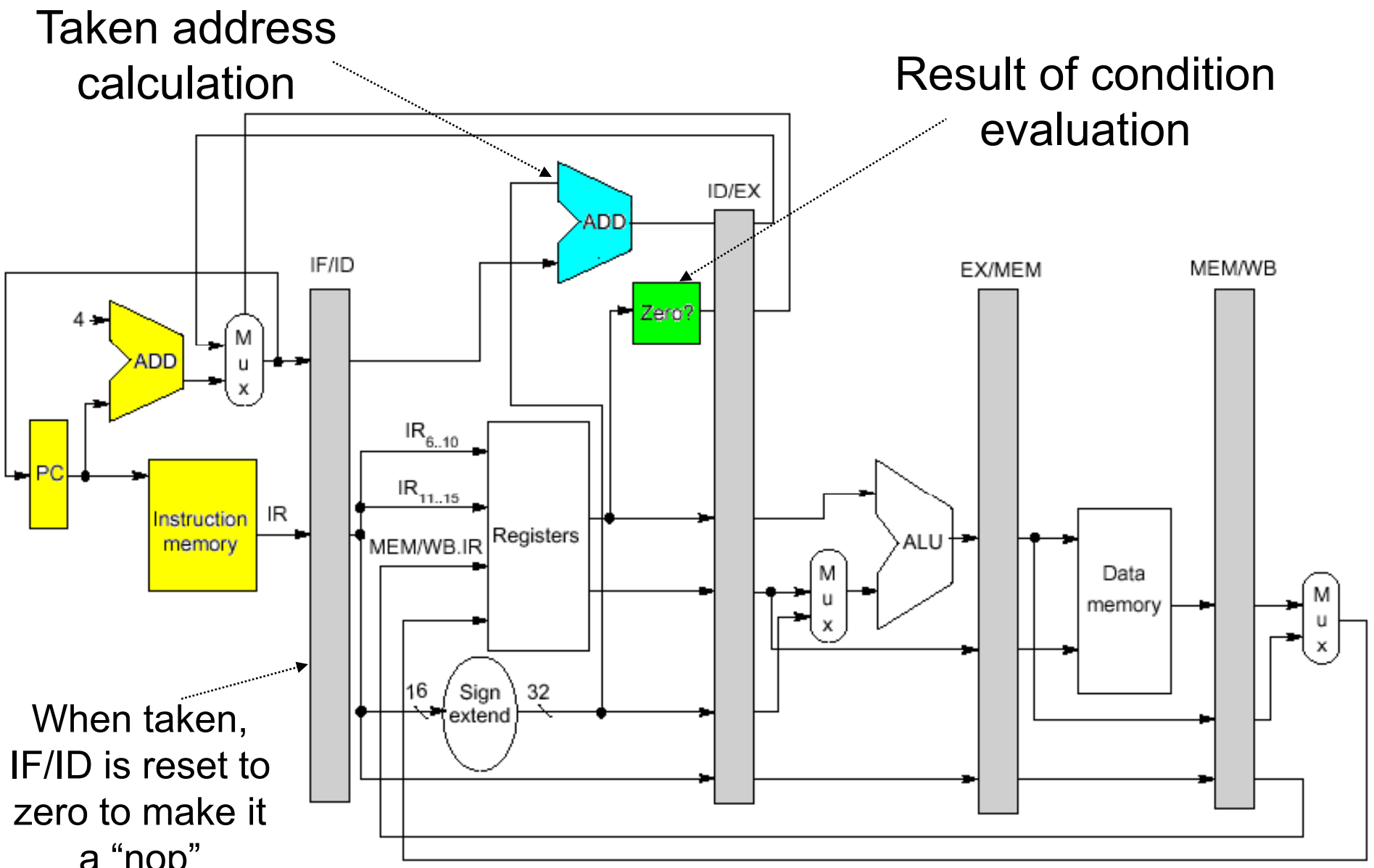
Control Hazard

- Control hazards are caused by the uncertainty of the execution path, branch taken or untaken
- If the branch is to be taken, the PC is not normally changed until the end of the MEM stage
- The simplest way to deal with control hazards is to stall the pipeline

Branch Instruction	IF	ID	EX	DM	WB					
Branch successor		IF	Stall	Stall	IF	ID	EX	DM	WB	ID
Branch successor + 1						IF	ID	EX	DM	WB
Branch successor + 2							IF	ID	EX	DM

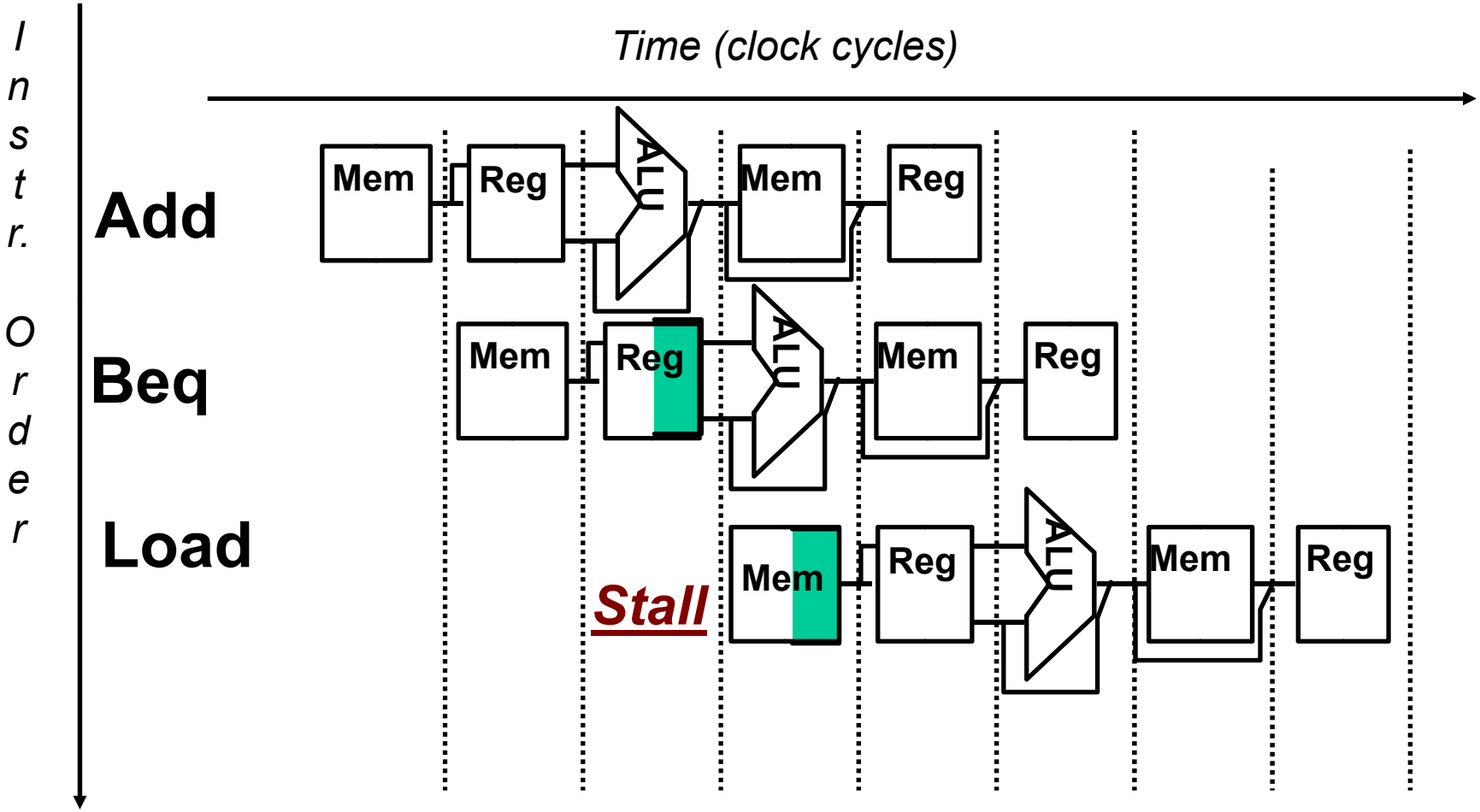
- Impact: 2 clock cycles wasted per branch instruction ⇒ slow
- Performance penalty can be limited by:
  - ➔ move up decision to 2nd stage by adding hardware to check registers as being read (*MIPS allows only comparison with zero in the condition*)
  - ➔ Compute the address of the branch target earlier in the pipeline

Enhanced Pipeline Datapath



Impact of Modifications

- Condition evaluated and taken address calculated at ID stage



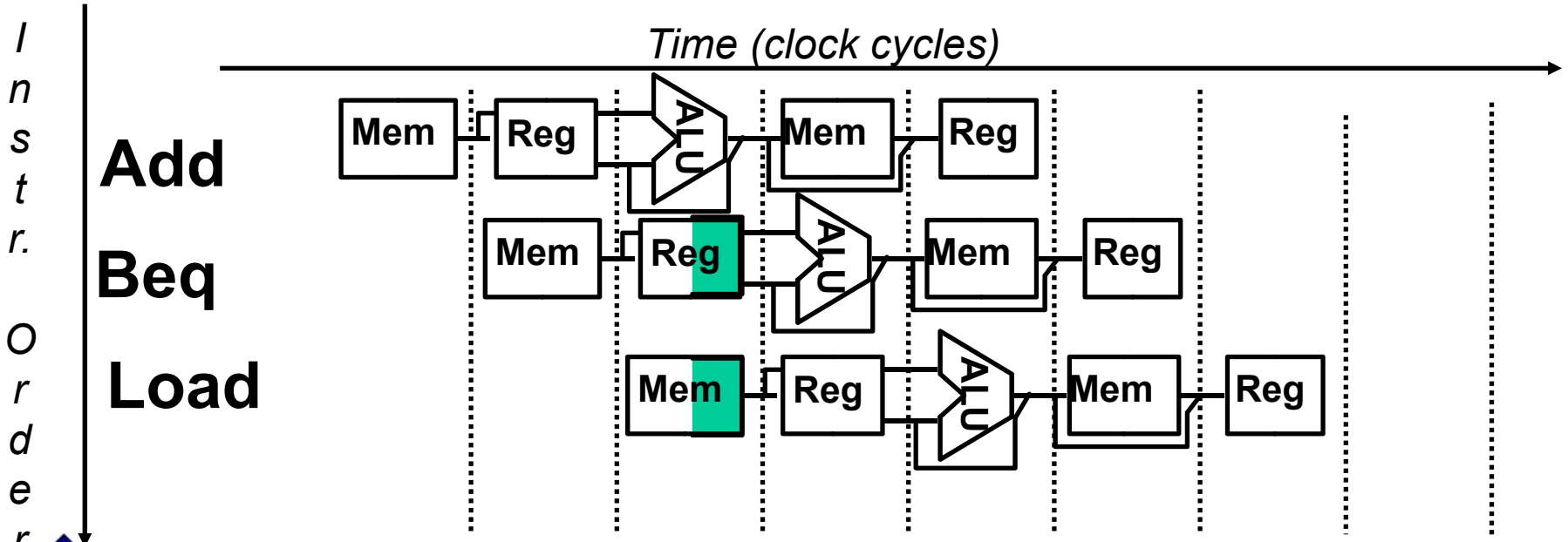
- 2 clock cycles per branch instr. (1 cycle wasted) ⇒ still slow

* Figure is courtesy of Dave Patterson

Control Hazard Solution (I)

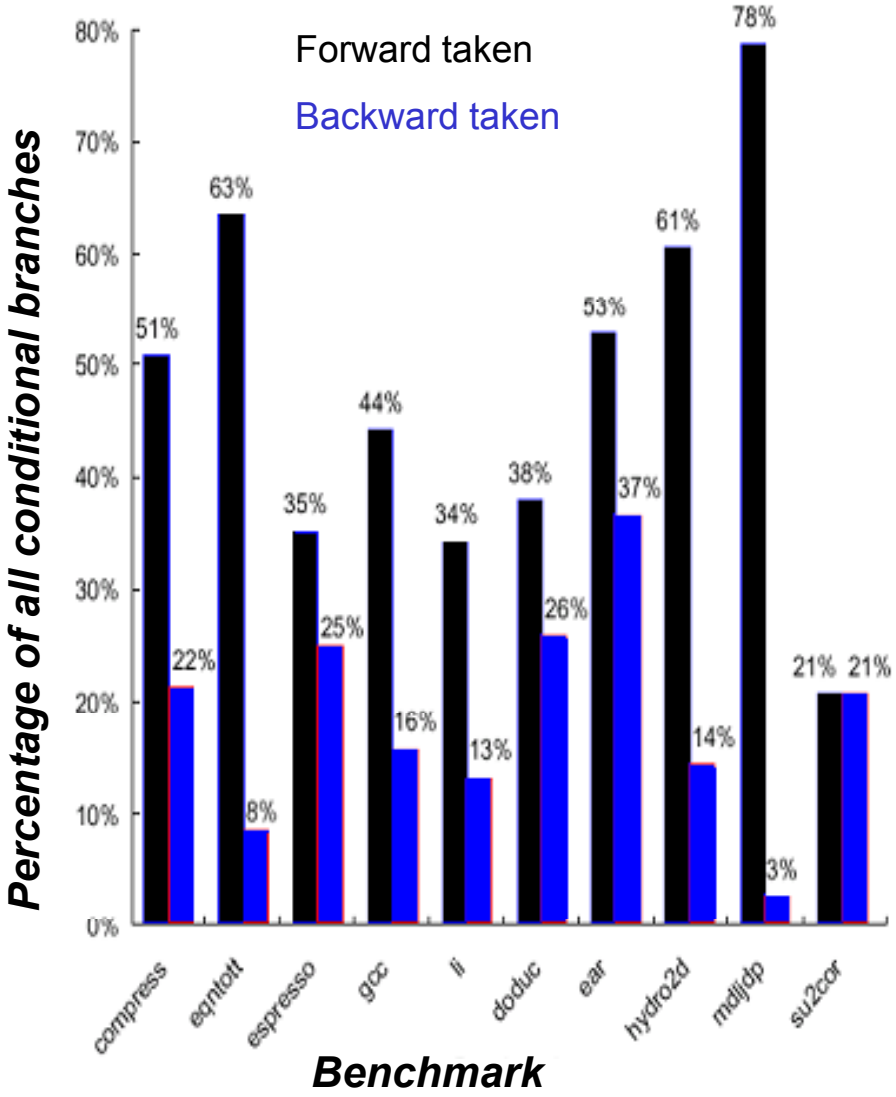
- Predict: guess one direction then back off if wrong
- Predict untaken: machine state must be updated to ensure correct semantics
- Impact: 1 clock cycles per branch instruction if right, 2 if wrong

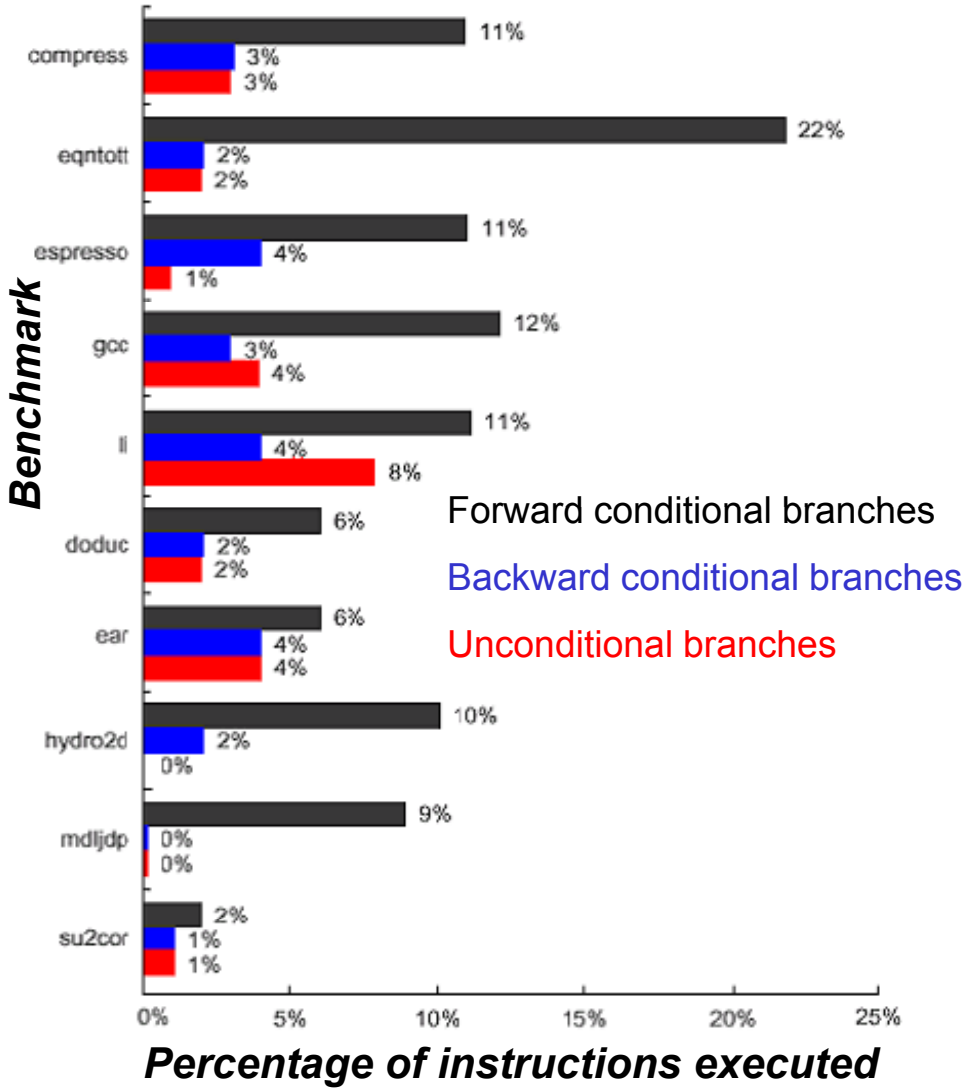
Taken Branch Instr.	IF	ID	EX	DM	WB				
Branch successor		IF	idle	idle	idle	idle			
Branch target			IF	ID	EX	DM	WB		
Branch target + 1				IF	ID	EX	DM	WB	
Branch target + 2					IF	ID	EX	DM	WB



* Figure is courtesy of Dave Patterson

Branch Behavior in Programs

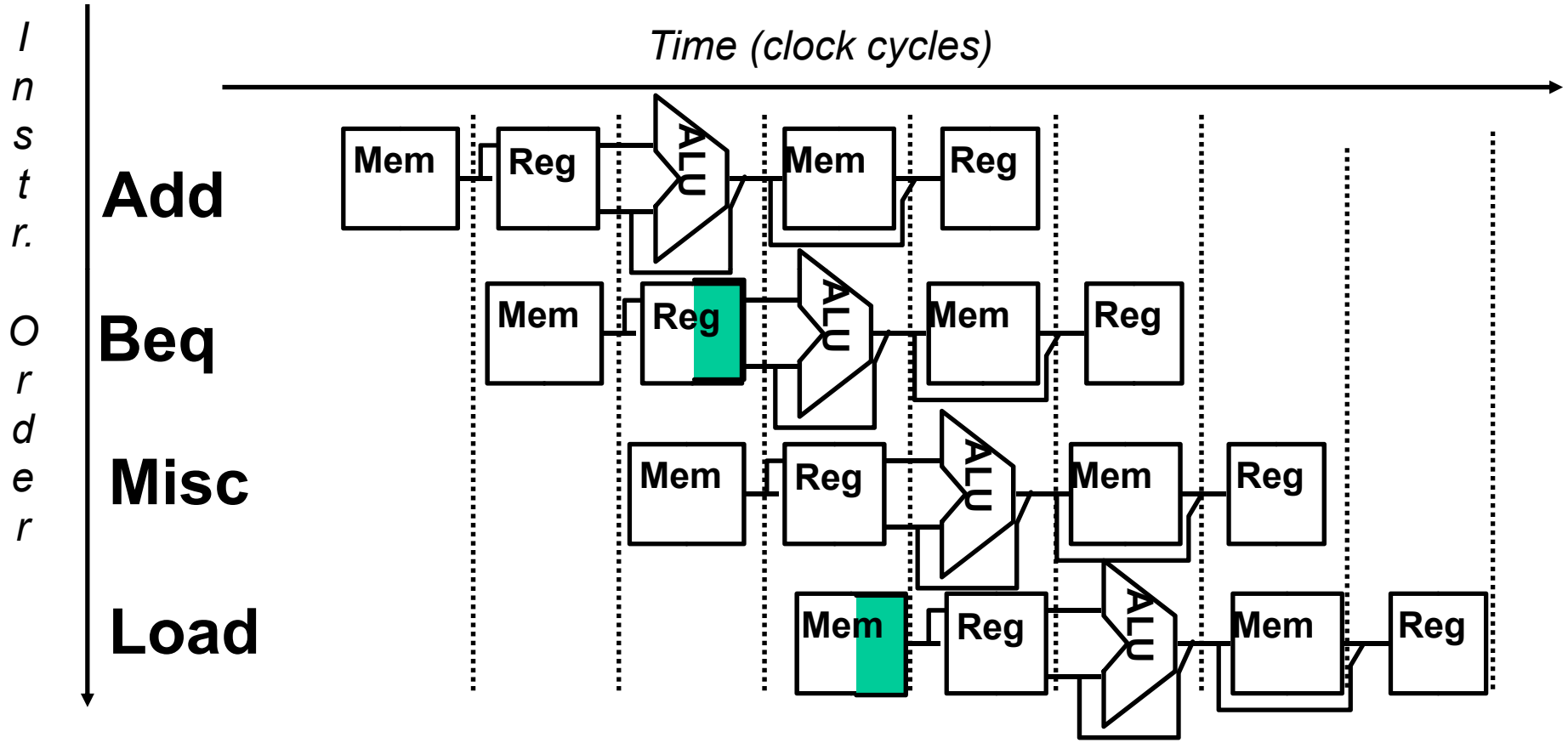




- 60% of forward and 15% of backward branches are taken (mainly loops)
- 67% of the branches (forward and backward) are taken on average

Control Hazard Solution (II)

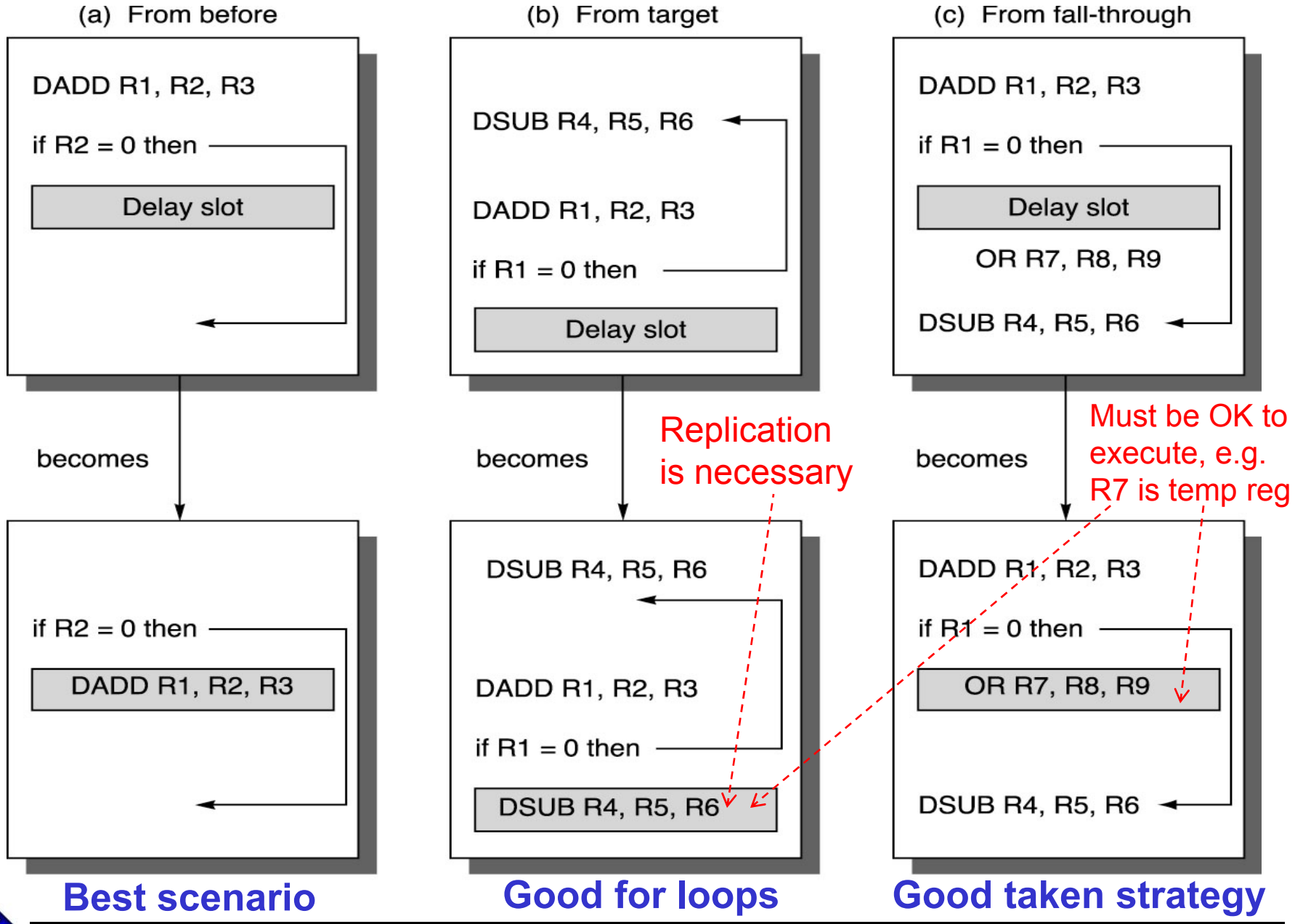
- Redefine branch behavior (takes place after next instr.) "delayed branch"
- Compiler optimization plays an essential role in filling delay slots



- Impact: 0 clock cycles per branch instruction if can find instructions to put in the delay "slot"

* Slide is courtesy of Dave Patterson

Scheduling Branch-Delay Slots

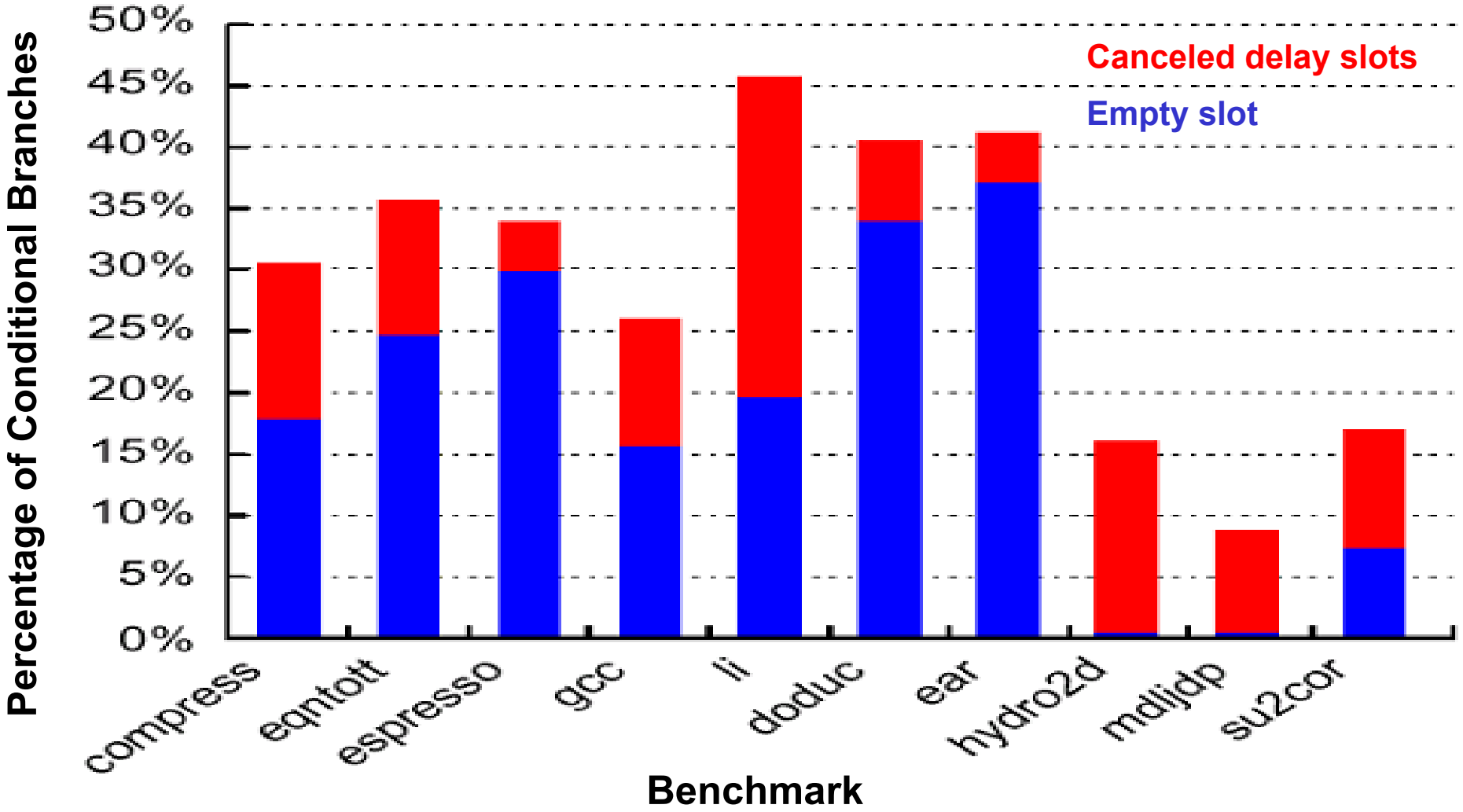


Branch-Delay Scheduling Requirements

Scheduling Strategy	Requirements	Improves performance when?
(a) From before	Branch must not depend on the rescheduled instructions	Always
(b) From target	Must be OK to execute rescheduled instructions if branch is not taken. May need to duplicate instructions.	When branch is taken. May enlarge programs if instructions are duplicated.
(c) From fall through	Must be okay to execute instructions if branch is taken.	When branch is not taken.

- The limitation on delayed-branch scheduling arise from:
  1. Restrictions on the instructions that are scheduled into the delay slots
  2. Ability to predict at compile-time whether a branch is likely to be taken
- To improve the ability of the compiler to fill branch delay slots with little undo penalty, a capability for cancellation (turning to no-op) is added
- Additional PC is needed to allow safe operation in case of interrupts

Performance of Branch-Delay



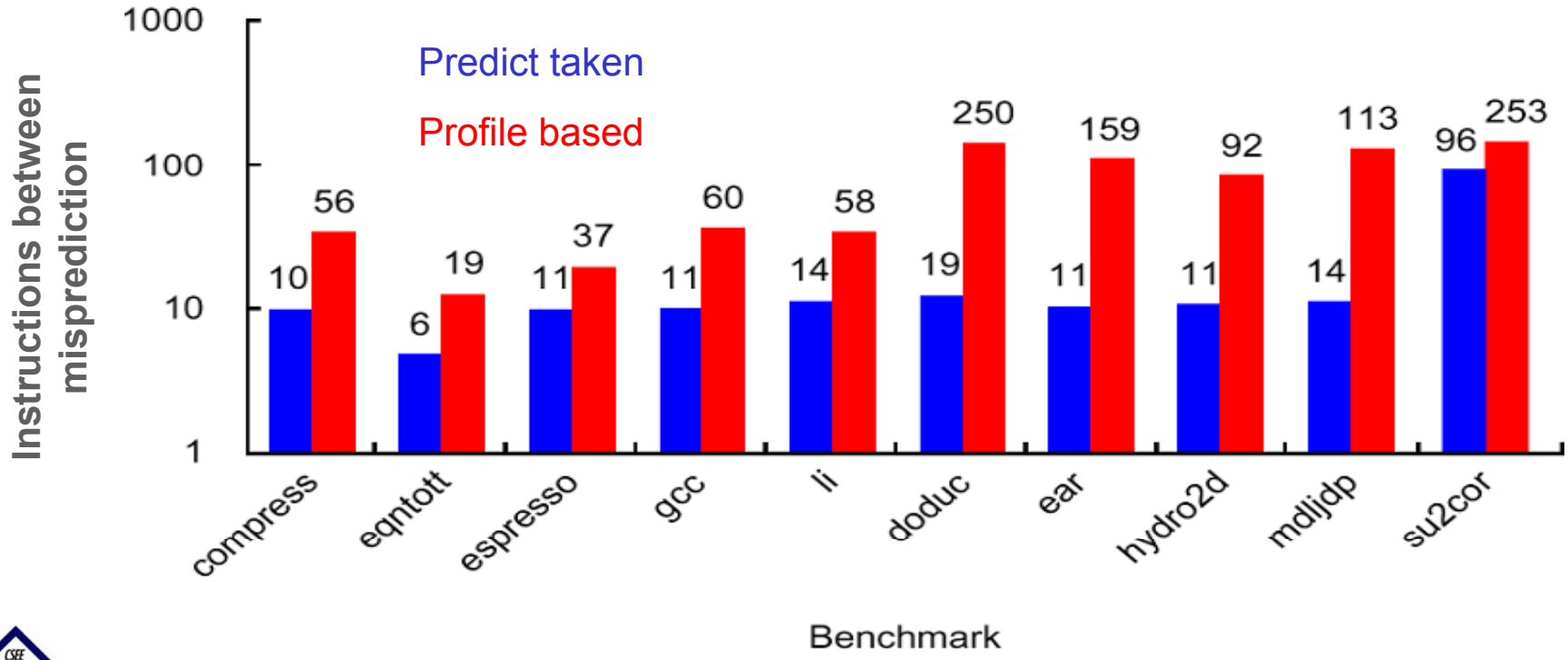
- On average 30% of the branch delay slots are wasted (integer programs are worse than floating point)

Static Branch Prediction

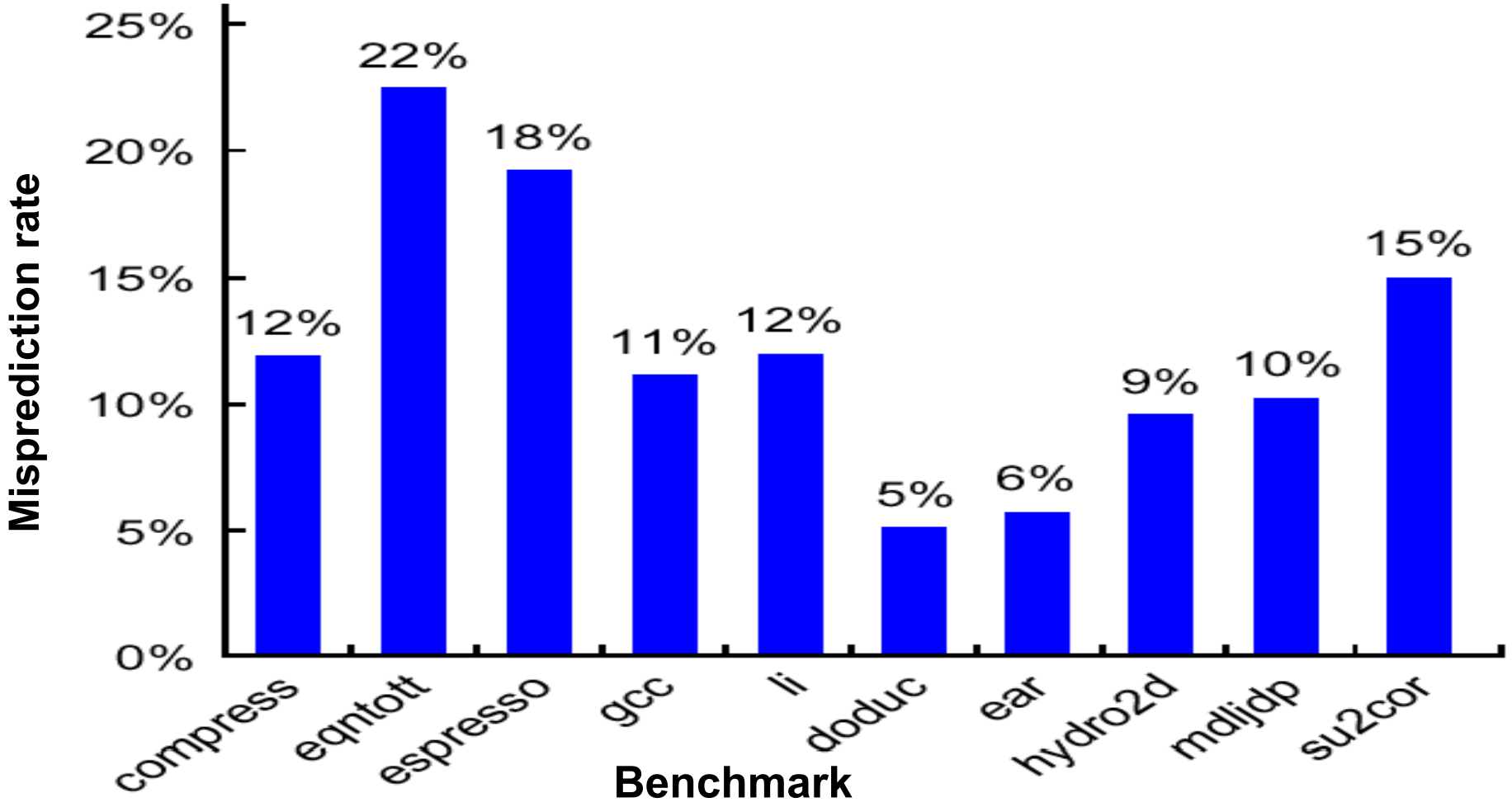
Examination of program behavior

- Assume branch is usually taken based on statistics but misprediction rate still range from 9%-59%
- Predict on branch direction forward/backward based on statistics and code generation convention

Profile information from earlier program runs

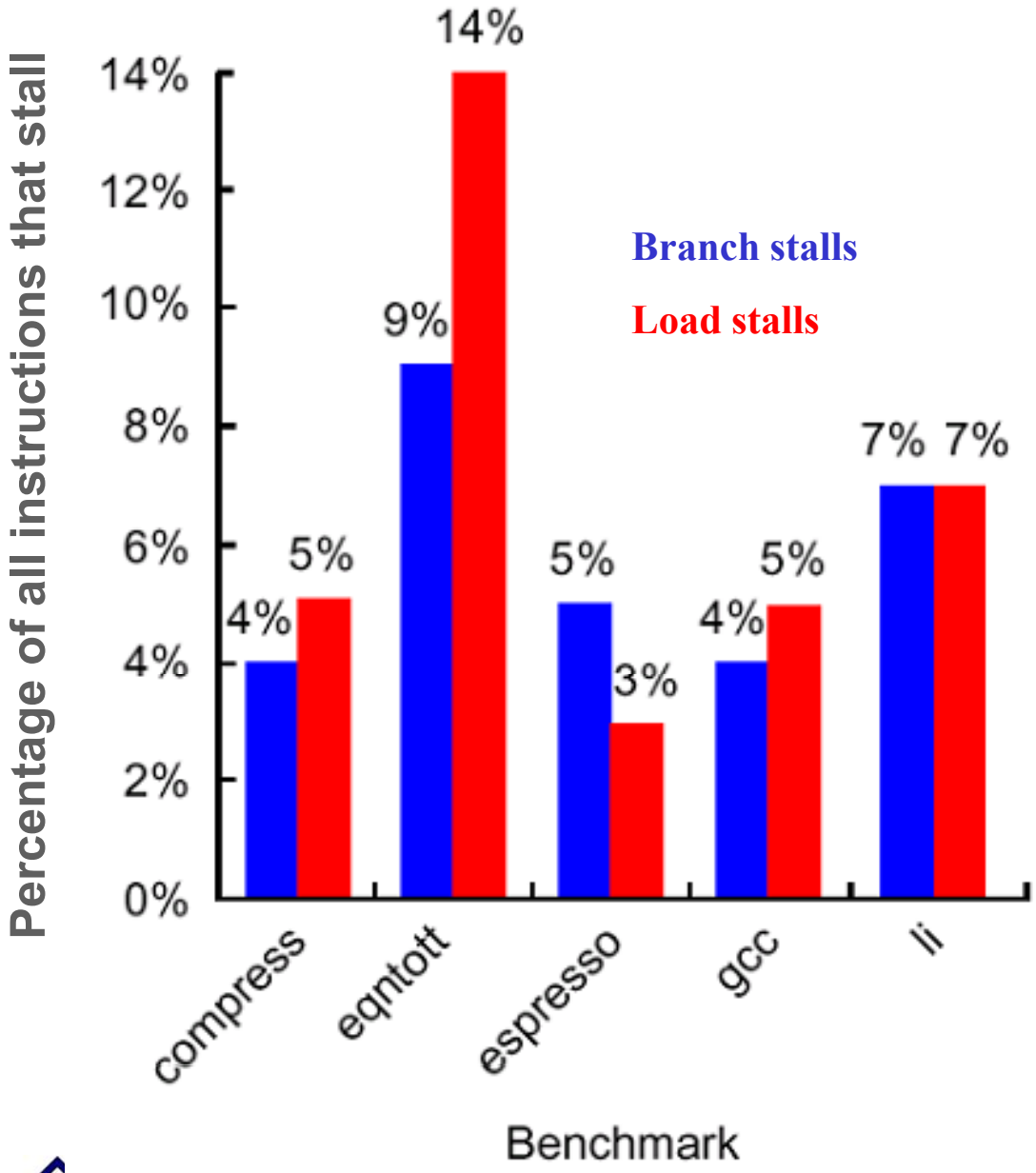


Profile-based Predictor Performance



- Misprediction rate varies widely but is generally better for FP programs
- Actual performance depends on both prediction accuracy & branch frequency

Performance of MIPS Integer Pipeline



- Performance is based on a perfect memory system with all cache hits
- Basic delayed branch with cancellation support and thus costing one delay cycle
- Integer programs exhibit an average of 0.06 branch stalls per instruction and 0.05 load stalls per instruction
- Average CPI from MIPS pipelining with perfect memory is 1.11
- SPECint92 performance can be enhanced by a factor of 5/1.11 = 4.5

Exception Types and Requirements

Famous Types of Exceptions

- I/O device request
- Breakpoint
- Integer arithmetic overflow
- FP arithmetic anomaly
- Page fault
- Misaligned memory accesses
- Memory-protection violation
- Undefined instruction
- Privilege violation
- Hardware and power failure

Requirements Characterization

- *Synchronous vs. asynchronous*
  - Async. Excep. are caused by I/O devices and allows completion of current instr.
- *User requested vs. coerced*
  - requested exceptions are predictable and easier to handle
- User *Maskable vs. unmaskable*
- *Within vs. between instructions*
  - Exceptions occurring within instructions are synchronous
  - It is harder to deal with exceptions that occur within instructions
- *Resume vs. terminate*
  - it is easier to implement exceptions that terminate program execution

Exception Categorization (Example)

Exception Type	Synchronous Vs Asynchronous	User req. vs coerced	User maskable Vs nonmaskable	Within Vs between instructions	Resume Vs terminate
I/O device request	Asynchronous	Coerced	nonmaskable	between	Resume
Invoke operating system	synchronous	User req.	nonmaskable	between	Resume
Tracing instruction execution	synchronous	User req.	maskable	between	Resume
Breakpoint	synchronous	User req.	maskable	between	Resume
Integer arithmetic overflow	synchronous	Coerced	maskable	Within	Resume
FP overflow or underflow	synchronous	Coerced	maskable	Within	Resume
Page fault	synchronous	Coerced	nonmaskable	Within	Resume
Misaligned memory access	synchronous	Coerced	maskable	Within	Resume
Memory protection violations	synchronous	Coerced	nonmaskable	Within	Resume
Using undefined instructions	synchronous	Coerced	nonmaskable	Within	terminate
Hardware malfunctions	Asynchronous	Coerced	nonmaskable	Within	terminate
Power failure	Asynchronous	Coerced	nonmaskable	Within	terminate

- Exceptions occurring within an instruction are the most difficult to handle since they require background saving of the program state
- Pipelines is called “restartable”, if it allows an instruction to be restarted after exception handling

Stopping & Restarting Execution

- Some exception, e.g. page fault, takes place while the instruction is in the MEM stage, requiring the instruction to be restarted
- When an exception occurs, the pipeline control can do the following:
  1. Force a trap instruction into the pipeline on the next IF
  2. Until the trap is taken, turn off all writes for the faulting instruction and all the instructions that follow
  3. After the exception-handling routine in the operating system receives control, it saves the PC of the faulting instruction
- When using delayed branching, it is impossible to recreate the state using a single PC since the instructions may not be sequentially related
- A pipeline that allows instructions before the faulting one to complete and those after it to restart, is said to have *precise exception*
- Ideally faulting instructions will not change the state of the machine before the exception occur, however if it does, exception handling gets complex
- Supporting precise exceptions simplifies the operating system interface and is required in machines that implements demand paging

Exceptions in MIPS

Pipeline Stage	Problem exceptions occurring
IF	Page fault on instruction fetch; misaligned memory access; memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory protection violation
WB	None

- Multiple exceptions might occur since multiple instructions are executing (LW followed by DIV might cause page fault and an arith. exceptions in same cycle)
- Exceptions can even occur out of order (e.g. a page fault of instr. memory can occur earlier than a page fault of data memory caused by the proceeding instruction in the pipeline)

Pipeline exceptions has to be handled in order of execution of faulting instructions not according to the time they occur

Precise Exception Handling

The MIPS Approach:

- Hardware posts all exceptions caused by a given instruction in a status vector associated with the instruction
- The exception status vector is carried along as the instruction goes down the pipeline
- Once an exception indication is set in the exception status vector, any control signal that may cause a data value to be written is turned off
- Upon entering the WB stage the exception status vector is checked and the exceptions, if any, will be handled according to the time they occurred
- Allowing an instruction to continue execution till the WB stage is not a problem since all write operations for that instruction will be disallowed

Notes:

- The MIPS machine design does not allow exception to occur at the WB stage
- All write operations in the MIPS pipeline are in late stages
- Machines that allow writing in early pipeline stages are difficult to handle since exceptions can occur after the machine state has been already changed

Instruction Set Complications

Early-Write Instructions

- An instruction that has a single write at the last stage of the pipeline, e.g. MIPS, can easily handle exceptions
- Machines that allow for multiple writes, e.g. VAX auto-increment, during instr. execution, usually require a capability to rollback the effect of an instruction
- Instructions that update the memory state during execution, e.g. string copy, temporary registers are saved and restored to allow instruction to continue

Branching mechanisms

- Code based branching set by non-branch instructions requires special care if an exception happen prior to executing the branch instruction

Multi-cycle operations

- In machines with widely different instruction cycles, an instruction can make multiple writes and cause complex data hazard, and thus exception become very hard to handle

If architects realize the relationship between instruction set design and pipelining, they can enable efficient pipelining

Conclusion

❑ *Summary*

➔ Control hazards

- Limiting the effect of control hazards via branch prediction and delay
- Static branch prediction techniques and performance
- Branch delay issues and performance

➔ Exceptions handling

- Categorizing of exception based on types and handling requirements
- Issues of stopping and restarting instructions in a pipeline
- Precise exception handling and the conditions that enable it
- Instructions set effects on complicating pipeline design

❑ *Next Lecture*

➔ Pipelining floating point operations

➔ An example pipeline: MIPS R4000

Reading assignment includes Appendix A.4 & A.5 in the textbook