Exploitation of instruction level parallelism

Computer Architecture

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Javier García Blas

ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid

Dynamic branch prediction

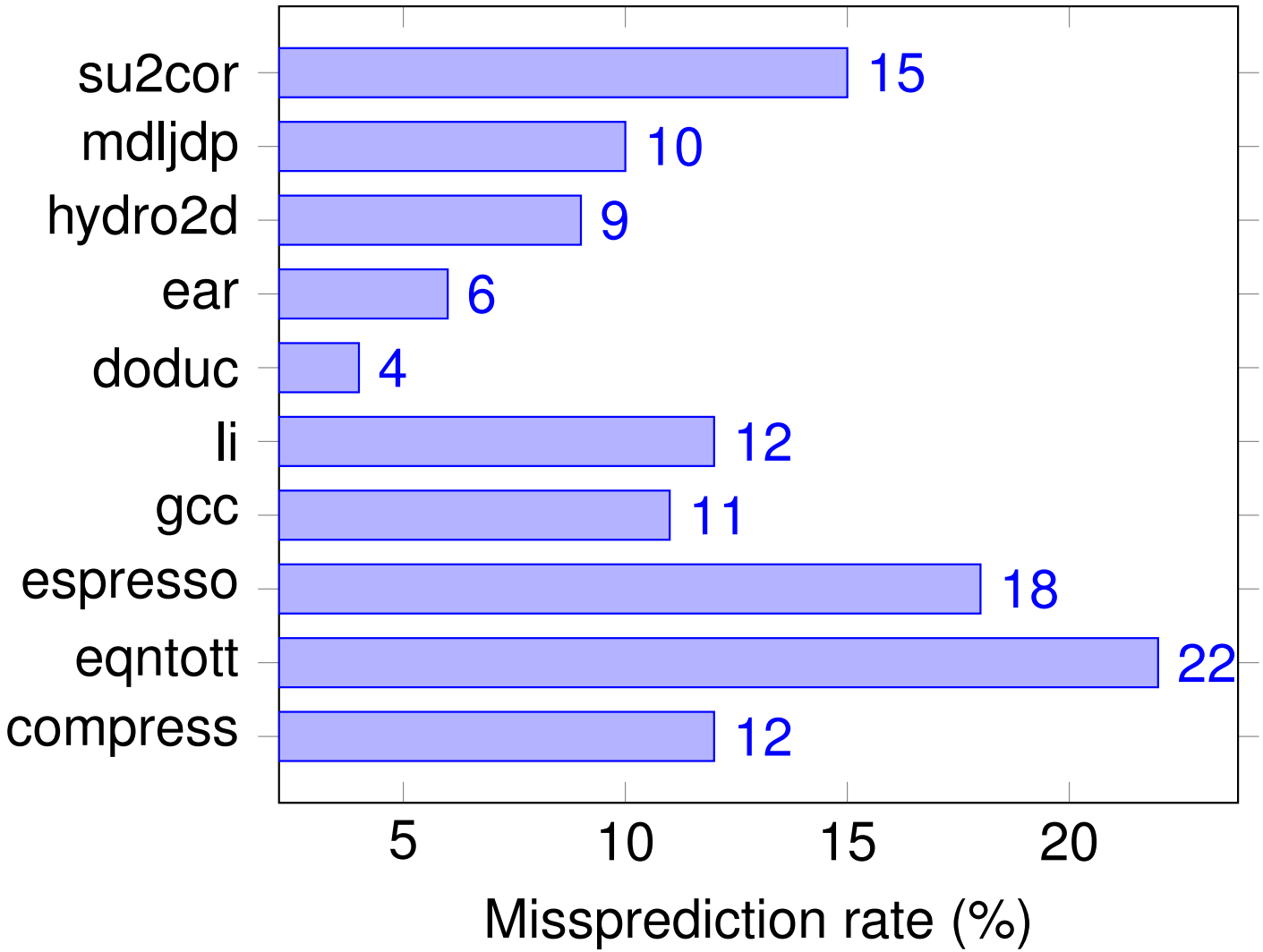
- Each conditional branch is **strongly biased**.
  - Either is taken most of the time,
  - or it is not taken most of the time.

- **Prediction based on execution profile**:
  - Run once to collect statistics.
  - Use the collected information to modify code and take advantage of information.

Predictions with execution profile

- SPEC92: Branch frequency 3% to 24%
- **Floating point**:
  - **Misprediction rate**.
    - **Average**: 9%.
    - **Standard deviation**: 4%.
- **Integer**:
  - **Misprediction rate**.
    - **Average**: 15%.
    - **Standard deviation**: 5%.

Predictions with execution profile



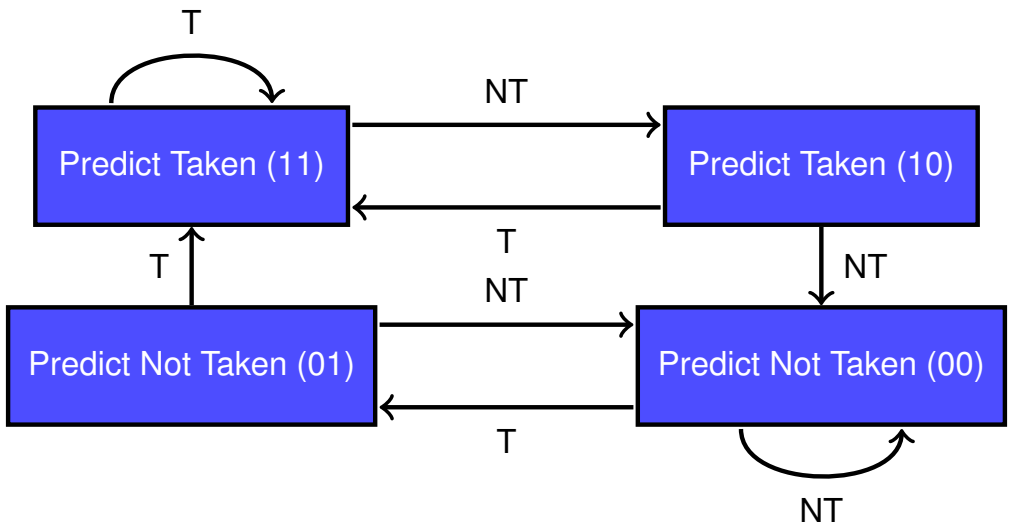
Dynamic prediction: BHT

- **Branch History Table**:
  - **Index**: Lower bits of address (**PC**).
  - **Value**: 1 bit (branch taken or not taken last time).

- **Effects**:
  - We don't know if the prediction is correct.
    - Might come from another instrucciton located at different address with same lower bits.
  - Number of lower bits implies size of the buffer
    - 10 lower bits $\Rightarrow$ 1024 entries.
  - If prediction fails bit is inverted
  - **Drawback**: A loop branch fails twice.
    - First and last iteration.

Dynamic prediction: BHT

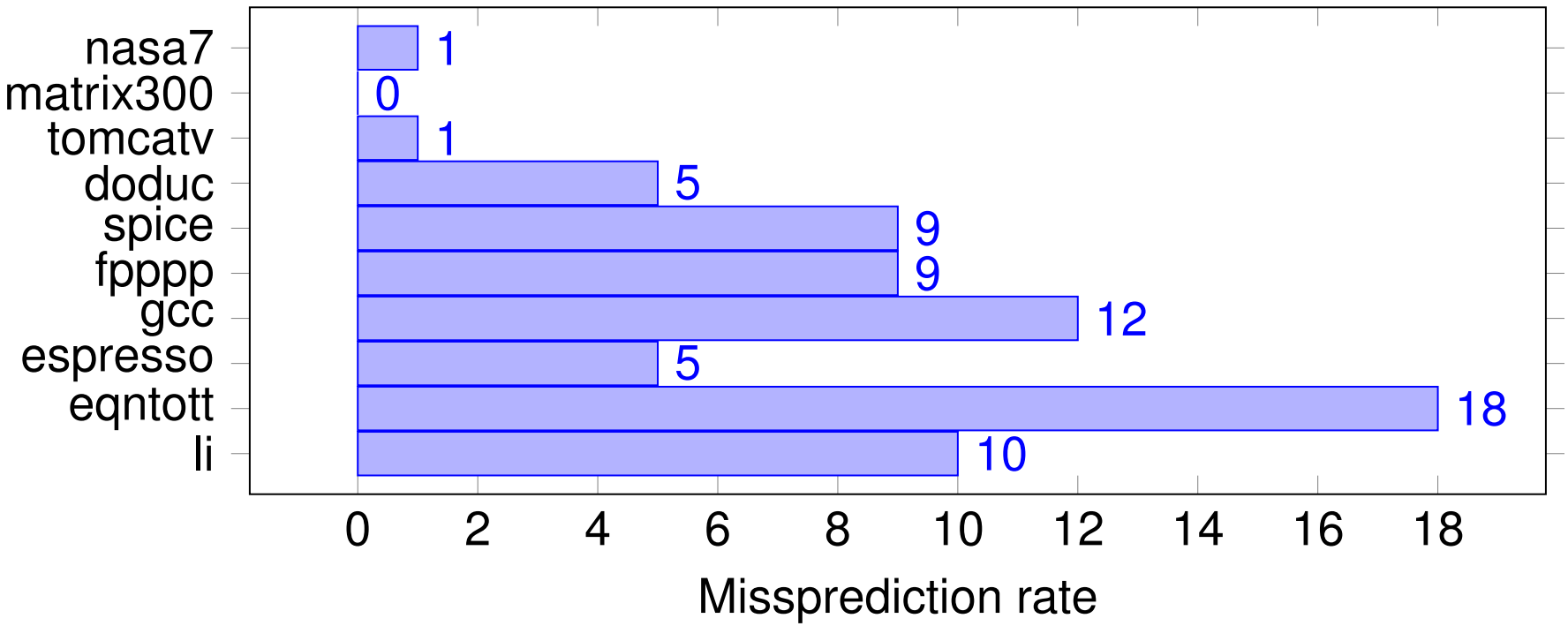
- **Branch History Table**:
  - **Index**: Lower bits of address (**PC**).
  - **Value**: 2 bits.
    - **00** and **01**: Predict not taken.
    - **10** and **11**: Predict taken.



- **Improvements**: Use more bits to improve precision.

BHT: Precision

- Misprediction rate:
  - Wrong prediction in branch outcome.
  - History of different branch in table entry.
- BHT results of 2 bits and 4K entries:



Dynamic branch prediction

- Why does branch prediction work?
  - Algorithms exhibit regularities.
  - Data structures exhibit regularities.

- Is dynamic prediction better than static prediction?
  - It looks like.
  - There is a small number of important branches in programs with dynamic behavior.

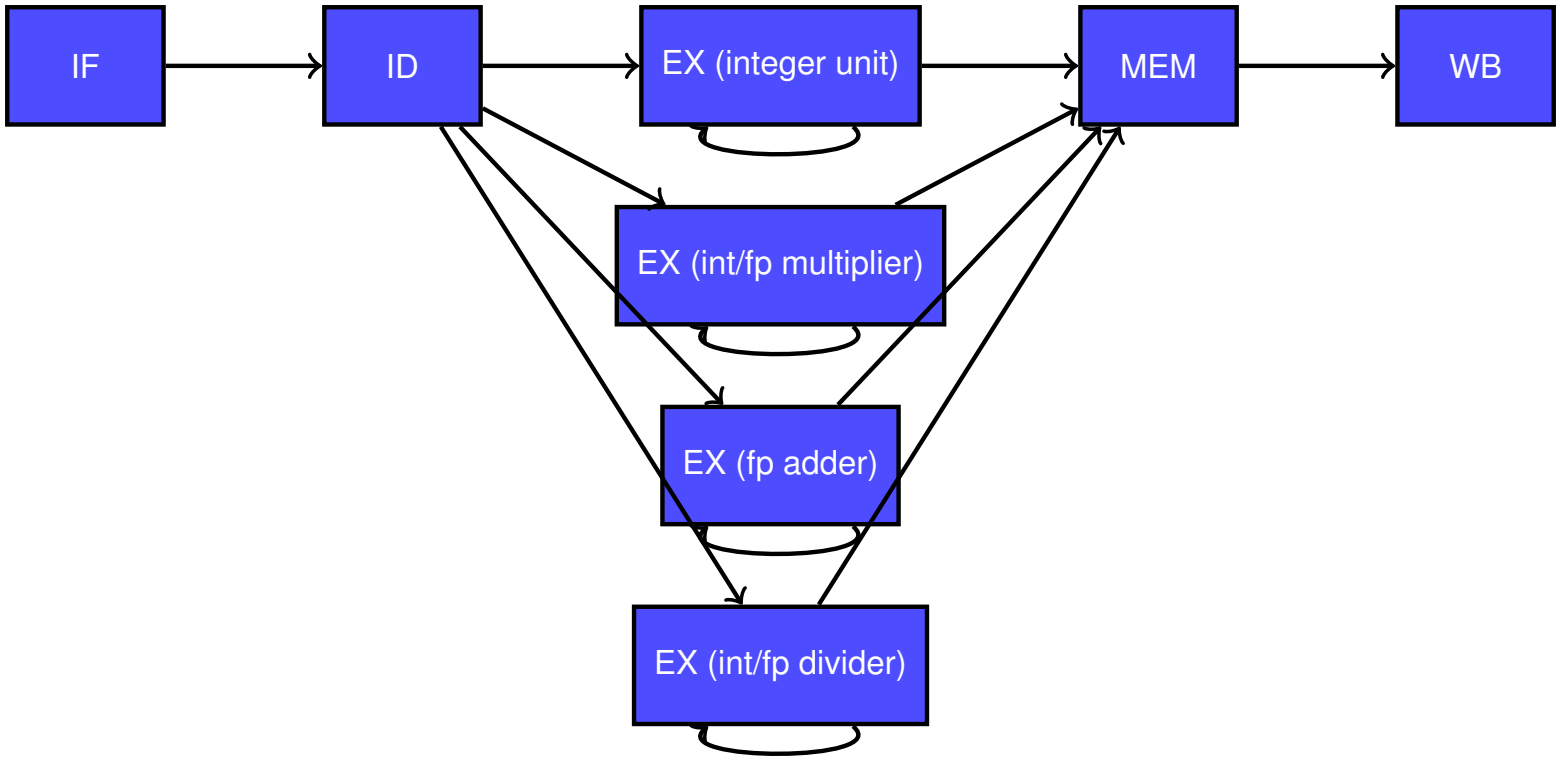
Floating point operations

- One-cycle floating point operations?
  - Having an extremely long clock cycle.
    - Impact on global performance.
  - Very complex FPU control logic.
    - Huge amount of resources for FP logic.

- **Alternative**: Floating point pipelining.
  - Execution stage may be repeated several times.
  - Multiple functional units in EX.
    - **Example**: Integer unit, FP and integer multiplier, FP adder, FP and integer divider.

Floating point pipeline

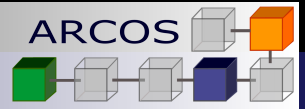
- EX stage now has a duration of more than 1 clock cycle.



Latency and initiation interval

- **Latency**: Number of cycles between instruction producing the result and instruction using the result.
- **Initiation interval**: Number of cycles between issue of two instructions using the same functional units

Operation	Latency	Initiation interval
Integer ALU	0	1
Loads	1	1
FP addition	3	1
FP multiplication	6	1
FP division	24	25



Taking advantage of ILP

- ILP directly applicable to basic blocks.
  - **Basic block**: sequence of instructions without branching.
  - **Typical** program in MIPS:
    - Basic block average size from 3 to 6 instructions.
    - Low ILP exploitation within block.
  - Need to exploit ILP across basic blocks.

Example

```
for (i=0;i<1000;i++) {
  x[i] = x[i] + y[i];
}
```

- **Loop level parallelism**.
  - Can be transformed to ILP.
  - By compiler or hardware.
- **Alternative**:
  - Vector instructions.
  - SIMD instructions in processor.

Scheduling and loop unrolling

- **Parallelism exploitation**:
  - Interleave execution of unrelated instructions.
  - Fill stalls with instructions.
  - Do not alter original program effects.

- Compiler can do this with detailed knowledge of the architecture.

ILP exploitation

Example

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

- Each iteration body is independent.

Latencies between instructions

Instruction producing result	Instruction using result	Latency (cycles)
FP ALU operation	FP ALU operation	3
FP ALU operation	Store double	2
Load double	FP ALU operation	1
Load double	Store double	0

Compiled code

- **R1** → Last array element.
- **F2** → Scalar **s**.
- **R2** → Precomputed so that **8(R2)** is the first element in array.

Assembler code

```
Loop:   L.D  F0, 0(R1)          ; F0 <- x[i]
        ADD.D F4, F0, F2        ; F4 <- F0 + s
        S.D  F4, 0(R1)          ; x[i] <- F4
        DADDUI R1, R1, #-8      ; i- -
        BNE R1, R2, Loop        ; Branch if R1!=R2
```

Stalls in execution

Original

```
Loop:   L.D F0, 0(R1)
        ADD.D F4, F0, F2
        S.D F4, 0(R1)
        DADDUI R1, R1, #-8
        BNE R1, R2, Loop
```

Stalls

```
Loop:   L.D F0, 0(R1)
        <stall>
        ADD.D F4, F0, F2
        <stall>
        <stall>
        S.D F4, 0(R1)
        DADDUI R1, R1, #-8
        <stall>
        BNE R1, R2, Loop
```

Loop scheduling

Original

```
Loop:   L.D F0, 0(R1)
        <stall>
        ADD.D F4, F0, F2
        <stall>
        <stall>
        S.D F4, 0(R1)
        DADDUI R1, R1, #-8
        <stall>
        BNE R1, R2, Loop
```

- 9 cycles per iteration.

Scheduled

```
Loop:   L.D F0, 0(R1)
        DADDUI R1, R1, #-8
        ADD.D F4, F0, F2
        <stall>
        <stall>
        S.D F4, 8(R1)
        BNE R1, R2, Loop
```

- 7 cycles per iteration.

Loop unrolling

- **Idea**:
  - Replicate loop body several times.
  - Adjust termination code.
  - Use different registers for each iteration replica to reduce dependencies.

- **Effect**:
  - Increase basic block length.
  - Increase use of available ILP.

Unrolling

Unrolling (x4)

```
Loop:   L.D F0, 0(R1)
        ADD.D F4, F0, F2
        S.D F4, 0(R1)
        L.D F6, -8(R1)
        ADD.D F8, F6, F2
        S.D F8, -8(R1)
        L.D F10, -16(R1)
```

Unrolling (x4)

```
        ADD.D F12, F10, F2
        S.D F12, -16(R1)
        L.D F14, -24(R1)
        ADD.D F16, F14, F2
        S.D F16, -24(R1)
        DADDUI R1, R1, #-32
        BNE R1, R2, Loop
```

- 4 iterations require more registers.
- This example assumes that array size is multiple of 4.

Stalls and unrolling

Unrolling (x4)

```
Loop:   L.D F0, 0(R1)
        <stall>
        ADD.D F4, F0, F2
        <stall>
        <stall>
        S.D F4, 0(R1)
        L.D F6, -8(R1)
        <stall>
        ADD.D F8, F6, F2
        <stall>
        <stall>
        S.D F8, -8(R1)
        L.D F10, -16(R1)
        <stall>
```

Unrolling (x4)

```
        ADD.D F12, F10, F2
        <stall>
        <stall>
        S.D F12, -16(R1)
        L.D F14, -24(R1)
        <stall>
        ADD.D F16, F14, F2
        <stall>
        <stall>
        S.D F16, -24(R1)
        DADDUI R1, R1, #-32
        <stall>
        BNE R1, R2, Loop
```

- 27 cycles for every 4 iterations $\rightarrow$ 6.75 cycles per iteration.

Scheduling and unrolling

Unrolling (x4)

```
Loop:   L.D   F0, 0(R1)
        L.D   F6, -8(R1)
        L.D   F10, -16(R1)
        L.D   F14, -24(R1)
        ADD.D F4, F0, F2
        ADD.D F8, F6, F2
        ADD.D F12, F10, F2
        ADD.D F16, F14, F2
        S.D   F4, 0(R1)
        S.D   F8, -8(R1)
        S.D   F12, -16(R1)
        DADDUI R1, R1, #-32
        S.D   F16, 8(R1)
        BNE   R1, R2, Loop
```

- Code reorganization.
  - Preserve dependencies.
  - Semantically equivalent.
  - **Goal**: Make use of *stalls*.
- Update of **R1** at enough distance from **BNE**.
- 14 cycles for every 4 iterations $\rightarrow$ 3.5 cycles per iteration.

Limits of loop unrolling

- **Improvement** is **decreased** with each additional unrolling.
  - Improvement limited to stalls removal.
  - Overhead amortized among iterations.
- **Increase** in code **size**.
  - May affect to instruction cache miss rate.
- **Pressure** on **register file**.
  - May generate shortage of registers.
  - Advantages are lost if there are not enough available registers.

Summary

- Stalls due to control hazards many be reduced with:
  - Compile time alternatives.
  - Run-time alternatives.
- Multi-cycle operations allow for shorter clock-cycles.
- Loop unrolling allows hiding stall latencies, but offers a limited improvement.

References

- **Computer Architecture. A Quantitative Approach**
  6th Ed.
  Hennessy and Patterson.
  Sections C.2, C.5, 3.1, 3.2

Exploitation of instruction level parallelism

Computer Architecture

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Javier García Blas

ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid