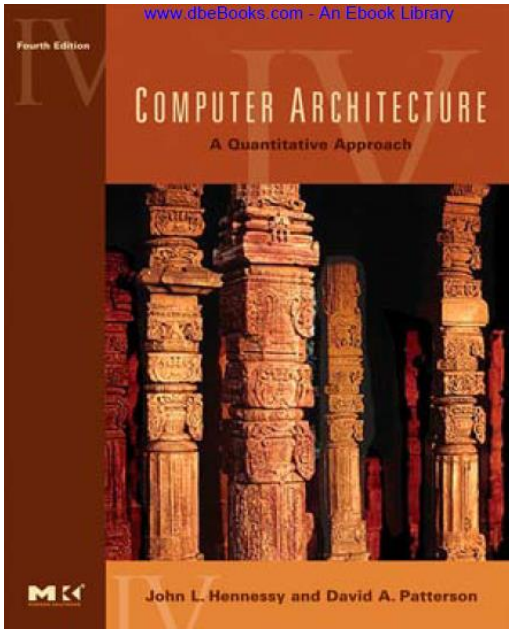
Instruction-Level Parallelism

compiler techniques and branch prediction

prepared and instructed by
Shmuel Wimer
Eng. Faculty, Bar-Ilan University



Concepts and Challenges

The potential overlap among instructions is called **instruction-level parallelism** (**ILP**).

Two approaches exploiting ILP:

- Hardware discovers and exploit the parallelism dynamically.
- Software finds parallelism, statically at compile time.

CPI for a pipelined processor:

Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls

Basic block: a straight-line code with no branches.

- Typical size between three to six instructions.
- Too small to exploit significant amount of parallelism.
- We must exploit ILP across multiple basic blocks.

Loop-level parallelism exploits parallelism among iterations of a loop. A completely parallel loop adding two 1000-element arrays:

```
for (i=1; i<=1000; i=i+1)
        x[i] = x[i] + y[i];
```

Within an iteration there is no opportunity for overlap, but every iteration can overlap with any other iteration.

The loop can be unrolled either statically by compiler or dynamically by hardware.

Vector processing is also possible. Supported in DSP, graphics, and multimedia applications.

Data Dependences and Hazards

If two instructions are **parallel**, they can be executed simultaneously in a pipeline without causing any stalls, assuming the pipeline has sufficient resources.

Two dependent instructions must be executed in order, but can often be partially overlapped.

Three types of dependences: **data dependences**, **name dependences**, and **control dependences**.

Instruction ***j*** is **data dependent** on instruction ***i*** if:

- ***i*** produces a result that may be used by ***j***, or
- ***j*** is data dependent on an instruction ***k***, and ***k*** is data dependent on ***i*** (transitivity).

The following loop increments a vector of values in memory by a scalar in register F2, starting at 0(R1), with the last element at 8(R2)).

```
Loop:   L.D     F0,0(R1)    ;F0=array element
        ADD.D   F4,F0,F2    ;add scalar in F2
        S.D     F4,0(R1)    ;store result
        DADDUI  R1,R1,#-8   ;decrement pointer 8 bytes
        BNE     R1,R2,LOOP  ;branch R1!=R2
```

The data dependences in this code sequence involve both floating-point and integer data.

Since between two data dependent instructions there is a chain of one or more data hazards, they cannot be executed simultaneously or completely overlap.

Data dependence conveys:

- the possibility of a hazard,
- the order in which results must be calculated, and
- an upper bound on how much parallelism can be exploited.

Detecting dependence of registers is straightforward.

- Register names are fixed in the instructions.

Dependences that flow through memory locations are more difficult to detect.

- Two addresses may refer to the same location but look different: For example, 100(R4) and 20(R6).
- The effective address of a load or store may change from one execution of the instruction to another (so that 100(R4) and 20(R6) may be different).

Name Dependences

A **name dependence** occurs when two instructions use the same register or memory location, called **name**, but there is no flow of data between the instructions. If ***i*** precedes ***j*** in program order:

Anti dependence between ***i*** and ***j*** occurs when ***j*** writes a register or memory location that ***i*** reads.

```
S.D    F4,0(R1)    ;store result
DADDUI R1,R1,#-8   ;decrement pointer 8 bytes
```

The original ordering must be preserved to ensure that ***i*** reads the correct value.

Output dependence occurs when ***i*** and ***j*** write the same register or memory location. Their ordering must be preserved to ensure proper value written by ***j***.

Name dependence is not a true dependence.

- The instructions involved can execute simultaneously or be reordered.
- The name (register # or memory location) is changed so the instructions do not conflict.

Register renaming can be more easily done.

- Done either statically by a compiler or dynamically by the hardware.

Data Hazards

A hazard is created whenever a dependence between instructions is close enough.

- **Program order** must be preserved.

The goal of both SW and HW techniques is to exploit parallelism by preserving program order **only where it affects the outcome of the program**.

- Detecting and avoiding hazards ensures that necessary program order is preserved.

Data hazards are classified depending on the order of read and write accesses in the instructions. Consider two instructions ***i*** and ***j***, with ***i*** preceding ***j***

The possible data hazards are:

RAW (**read after write**). ***j*** tries to read a source before ***i*** writes it.

- The most common, corresponding to a true data dependence.
- Program order must be preserved.

WAW (**write after write**). ***j*** tries to write an operand before it is written by ***i***.

- Writes are performed in the wrong order, leaving the value written by ***i*** rather than by ***j***.
- Corresponds to an output dependence.
- Present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled.

WAR (**write after read**). ***j*** tries to write a destination before it is read by ***i***, so ***i*** incorrectly gets the new value.

- Arises from **anti dependence**.
- Cannot occur in most static issue pipelines because all reads are early (in ID) and all writes are late (in WB).
- Occurs when there are some instructions that write results early in the pipeline and other instructions that read a source late in the pipeline.
- Occurs also when instructions are reordered.

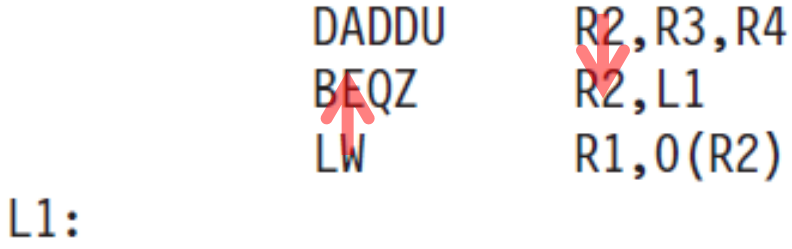
Control Dependences

A **control dependence** determines the ordering of ***i*** with respect to a branch so that ***i*** is executed in correct order and only when it should be.

There are two constraints imposed by control dependences:

- An instruction that is control dependent on a branch cannot be moved **before** the branch so that its execution **is no longer controlled** by the branch.
- An instruction that is **not** control dependent on a branch cannot be moved **after** the branch so that its execution **is controlled** by the branch.

Consider this code:



If we do not maintain the **data dependence** involving R2, the result of the program can be changed.

If we ignore the **control dependence** and move the load before the branch, the load may cause a memory protection exception. (why?)

It is not **data dependence** preventing interchanging the BEQZ and the LW; it is only the **control dependence**.

Compiler Techniques for Exposing ILP

Pipeline is kept full by finding sequences of unrelated instructions that can be overlapped in the pipeline.

To **avoid stall**, a dependent instruction must be **separated** from the source by a distance in clock cycles equal to the **pipeline latency** of that source.

Example: Latencies of FP operations

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

Code adding scalar to vector:

```
for (i=1000; i>0; i=i–1)
    x[i] = x[i] + s;
```

Straightforward MIPS assembly code:

```
Loop:   L.D     F0,0(R1)    ;F0=array element
        ADD.D   F4,F0,F2    ;add scalar in F2
        S.D     F4,0(R1)    ;store result
        DADDUI  R1,R1,#-8   ;decrement pointer
                            ;8 bytes (per DW)
        BNE     R1,R2,Loop  ;branch R1!=R2
```

R1 is initially the **top** element address in the array.

F2 contains the scalar value s.

R2 is pre computed, so that 8(R2) is the array **bottom**.

Without any scheduling the loop takes 9 cycles:

```
Loop:  L.D      F0,0(R1)      1
       stall                  2
       ADD.D    F4,F0,F2      3
       stall                  4
       stall                  5
       S.D      F4,0(R1)      6
       DADDUI   R1,R1,#-8     7
       stall                  8
       BNE      R1,R2,Loop    9
```

Scheduling the loop obtains only two stalls, taking 7 cycles:

```
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
```

The actual work on the array is just $3/7$ cycles (load, add, and store). The other 4 are loop overhead. Their elimination requires more operations relative to the overhead.

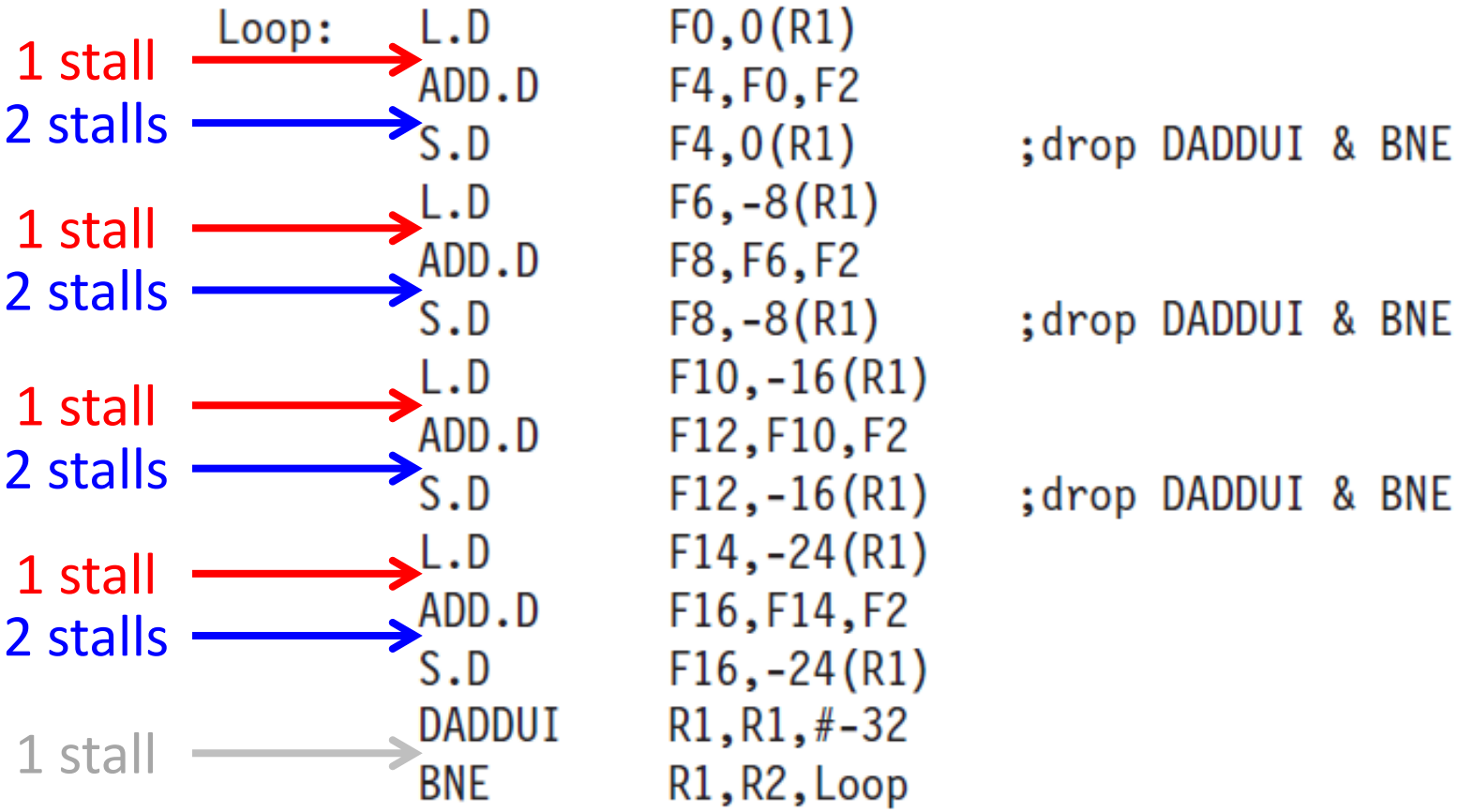
Loop unrolling replicating the loop body multiple times.

- Adjustment of the loop termination code is required.
- Used also to improve scheduling.

Instruction replication is insufficient. Different registers for each replication are required.

Required number of registers increases.

Unrolled code (not rescheduled)



Stalls are still there.

Run in 27 clock cycles, 6.75 per block.

Unrolled and rescheduled code

```
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)
        DADDUI  R1,R1,#-32
        S.D     F12,16(R1)
        S.D     F16,8(R1)
        BNE     R1,R2,Loop
```

No stalls are required!

Execution dropped to 14 clock cycles, 3.5 per block.

Compared with 9 per block before unrolling or scheduling and 7 when scheduled but not unrolled.

Is it a hazard?

Problem: The number of loop iterations n is usually unknown. We would like to unroll the loop to make k copies of its body.

Two consecutive loops are generated Instead.

The first executes $n \bmod k$ times and has a body that is the original loop.

The second is the unrolled body surrounded by an outer loop that iterates $\lfloor n/k \rfloor$ times.

For large n, most of the execution time will be spent in the unrolled loop body.

Branch Prediction

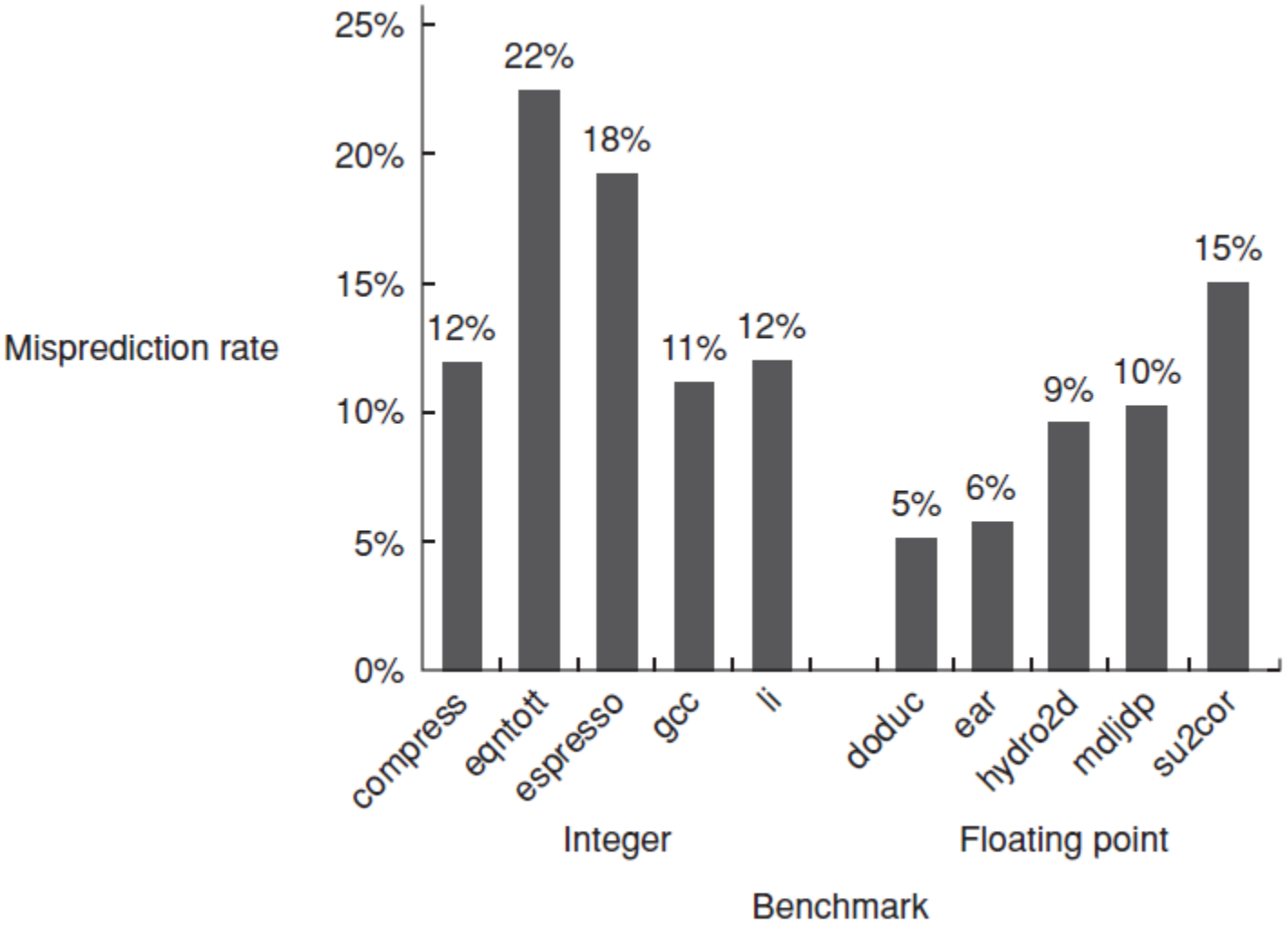
performance losses can be reduced by predicting how branches will behave.

Branch prediction (**BP**) can be done **statically** at compilation (SW) and **dynamically** at execution time (HW).

The simplest static scheme is to predict a branch as **taken**. Misprediction equal to the **untaken** frequency (34% for the SPEC benchmark).

BP based on profiling is more accurate.

Misprediction on SPEC92 for a profile-based predictor



Dynamic Branch Prediction

The simplest is a **BP buffer**, a small 1-bit memory indexed by the LSBs of the address of the branch instruction (no tags).

BP may have been put there by another branch that has the same LSBs address bits!

Useful only to reduce the branch delay (stalls) when it is longer than the time to compute the possible target PC address (e.g. $if\ \sin(x) < 0$).

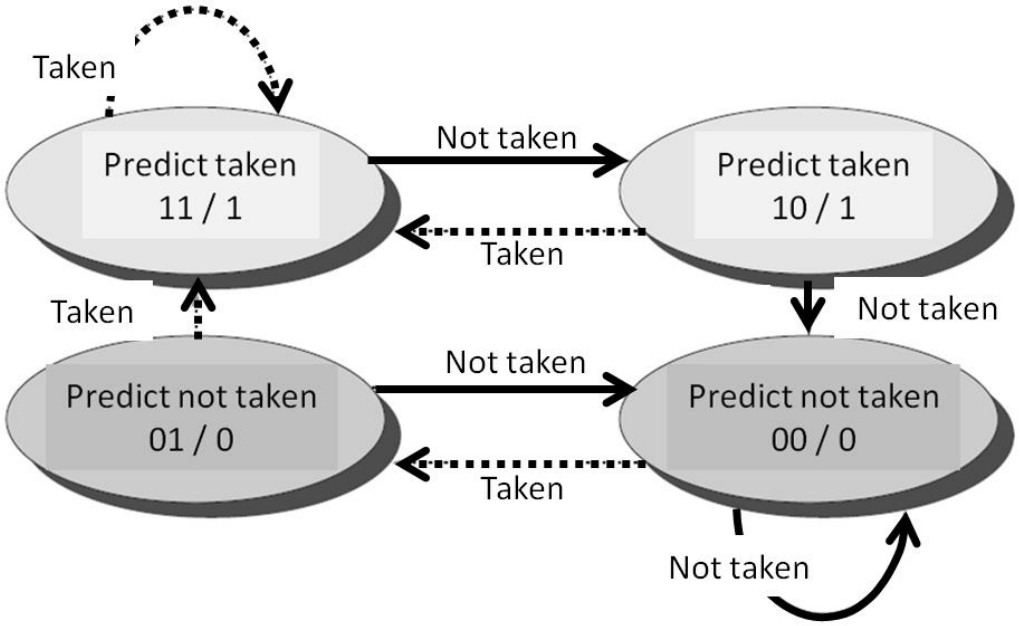
Fetching begins in the predicted direction. If it was wrong, the BP bit is inverted and stored back.

Problem: Even if **almost always** taken, we will likely predict incorrectly twice. (why?)

Example: Consider a certain loop. Upon exiting the loop a miss prediction occurs. Re-entry of that loop will cause another miss prediction.

Solution:

saturation counter.



It is a 2-bit saturation counter. It must miss twice before it is changed. Such counter is stored at every entry of the BP buffer.

It can be implemented as a special cache, read at IF (why?), or by adding two special bits to the I-cache.

An n-bit counter is also possible. When the counter is less than half, $\text{counter} \leq 2^{n-1} - 1$, not taken is predicted; otherwise, taken is predicted.

The counter is then updated according to the real branch decision.

2-bit do almost as well, thus used by most systems.

Correlating Branch Predictors

2-bit BP uses only the recent behavior of a single branch for a decision.

Accuracy can be improved if the recent behavior of **other** branches are considered.

Consider the code:

```
if (aa==2)
        aa=0;
if (bb==2)
        bb=0;
if (aa!=bb) {
```

Let aa and bb be assigned to registers R1 and R2, and label the three branches b1, b2, and b3. The compiler generates the typical MIPS code:

```
        DADDIU    R3,R1,#-2
        BNEZ      R3,L1          ;branch b1   (aa!=2)
        DADD      R1,R0,R0       ;aa=0
L1:     DADDIU    R3,R2,#-2
        BNEZ      R3,L2          ;branch b2   (bb!=2)
        DADD      R2,R0,R0       ;bb=0
L2:     DSUBU     R3,R1,R2       ;R3=aa-bb
        BEQZ      R3,L3          ;branch b3   (aa==bb)
```

The behavior of b3 is correlated with that of b1 and b2.

A predictor using only the behavior of a single branch to predict its outcome is blind of this behavior.

Correlating or **two-level predictors** add information about the most recent branches to decide on a branch.

For example, a $(1,2)$ BP uses the behavior of the last branch to choose from among a pair of 2-bit BPs in predicting the a particular branch.

An (m, n) BP uses the last m branches to choose from 2^m branch predictors, each of which is an n-bit predictor (counter) for a single branch.

More accurate than 2-bit and requires simple HW.

The global history of the most recent m branches is recorded in an m-bit shift register.

A (2^{m+r})-size BP buffer is indexed by $(m + r)$-bit using the r LSBs branch address and m-bit recent history.

For example, in a $(2,2)$ BP buffer with 64 total entries, the 6-bit index of a 64 entries is formed by the 4 LSBs of the branch address plus 2 global bits obtained from the two most recent branches behavior.

For a fair comparison of the performance of BPs, the same number of state bits are used.

The number of bits in an (m, n) predictor is:
$2^m \times n \times$ # BP entries selected by the branch address $= 2^{m+r} \times n$.

A 2-bit predictor w/o global history is a $(0,2)$ predictor.

Example: How many bits are in the $(0,2)$ BP with 4K entries? How many entries are in a $(2,2)$ predictor with the same number of bits?

A 4K-entries $(0,2)$ BP has $2^0 \times 2 \times 4\text{K}{=}8\text{K}$ bits.

A $(2,2)$ BP having a total of 8K bits satisfies:

$2^m \times n \times$ # BP entries selected by the branch address.

$2^2 \times 2 \times$ # BP entries selected by the branch address $= 8\text{K}$ bits.

The # of prediction entries is therefore 1K.


4096 entries:
2 bits per entry
Unlimited entries:
2 bits per entry
1024 entries:
(2,2)
not much improvement
significant improvement
Frequency of mispredictions
0%
2%
4%
6%
8%
10%
12%
14%
16%
18%
nasa7
matrix300
tomcatv
doduc
spice
fpppp
gcc
espresso
eqntott
li
1%
0%
1%
0%
0%
0%
1%
0%
1%
5%
5%
5%
9%
9%
5%
9%
9%
5%
12%
11%
11%
5%
5%
4%
18%
18%
6%
10%
10%
5%

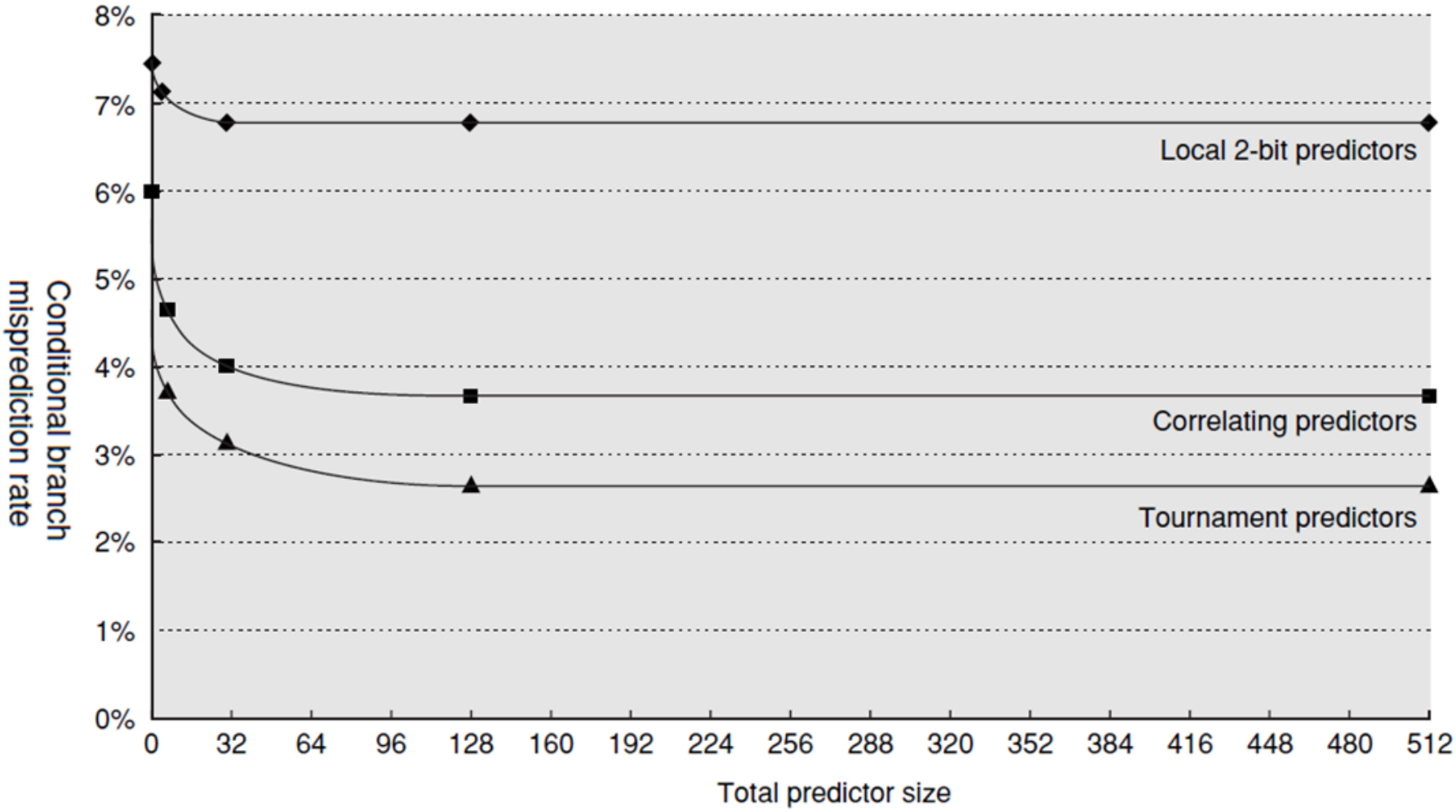
Tournament Predictors

Tournament predictors combine predictors based on global and local information.

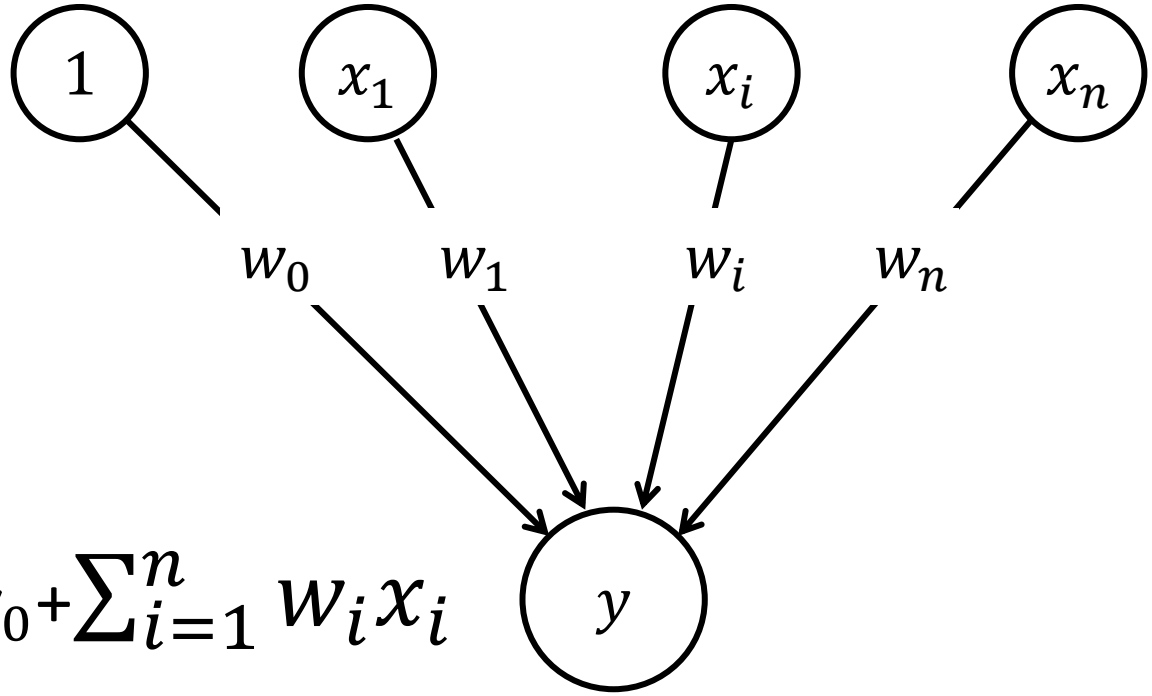
They achieve better accuracy and effectively use very large numbers of prediction bits.

Tournament BPs use a 2-bit saturating counter per branch to select between two different BP (local, global), based on which was most effective in recent predictions.

As in a simple 2-bit predictor, the saturating counter requires two mispredictions before changing the identity of the preferred BP.


8%
7%
6%
5%
4%
3%
2%
1%
0%
Conditional branch misprediction rate
Local 2-bit predictors
Correlating predictors
Tournament predictors
0
32
64
96
128
160
192
224
256
288
320
352
384
416
448
480
512
Total predictor size

Perceptron-Based Branch Predictor



$$y = w_0 + \sum_{i=1}^{n} w_i x_i$$

x_i are ± 1 bits of global recent n-bit branch history.

$t = \pm 1$ is the real branch outcome (taken, not taken).

Weights update:

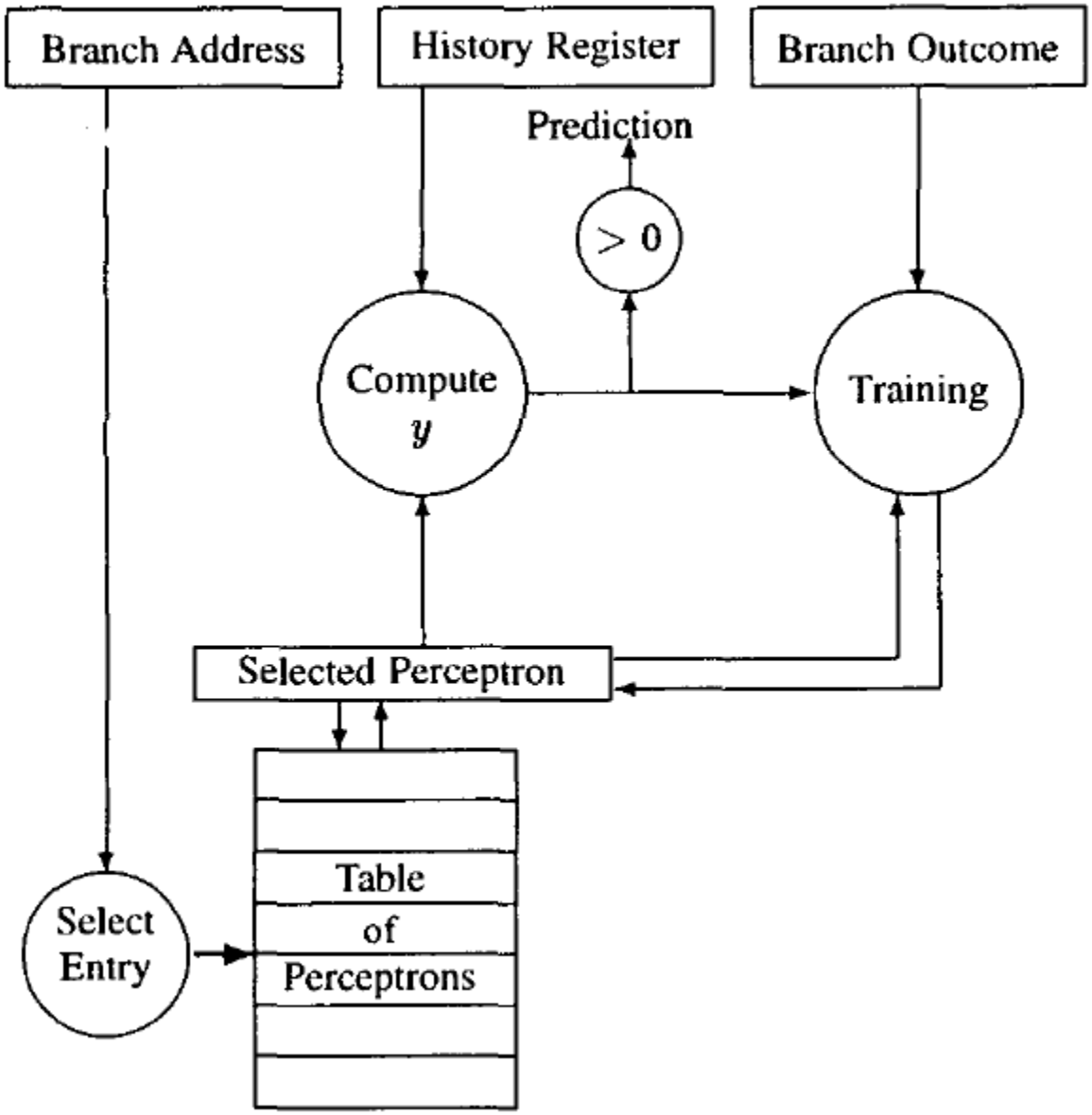
$\boldsymbol{\theta}$ training threshold

$$\text{If } \big((\text{sign}(y) \neq t\,)||(|y| < \boldsymbol{\theta})\big)\,\{$$
$$w_i = w_i + tx_i\,,\ 1 \leq i \leq n;$$
$$\}$$

Since $t = \pm 1$ and $x_i = \pm 1$, w_i increases when the branch outcome agrees with x_i (positive correlation) and decreases on disagreement (negative correlation).

Long lasting positive or negative correlation yield large weights, hence large influence on the prediction.

Weak correlation maintains the weight close to 0, contributing a little to the output of the perceptron.


Branch Address
History Register
Branch Outcome
Prediction
> 0
Compute
y
Training
Selected Perceptron
Table
of
Perceptrons
Select
Entry

History length $n = 10 - 100$.

The longer the more accurate predictions, but more memory for weights (per table entry).