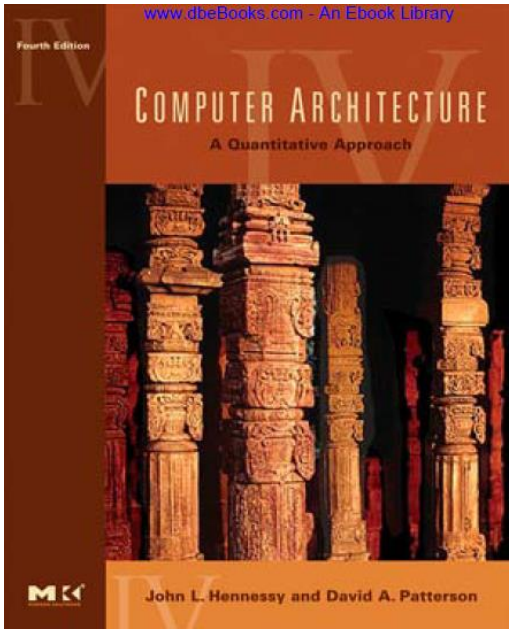
Instruction-Level Parallelism

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 3-6 instructions.
- Too small to exploit significant amount of parallelism.
- Exploit ILP across multiple basic blocks.

Loop-level parallelism Completely parallel loop adding two 1000-element arrays:

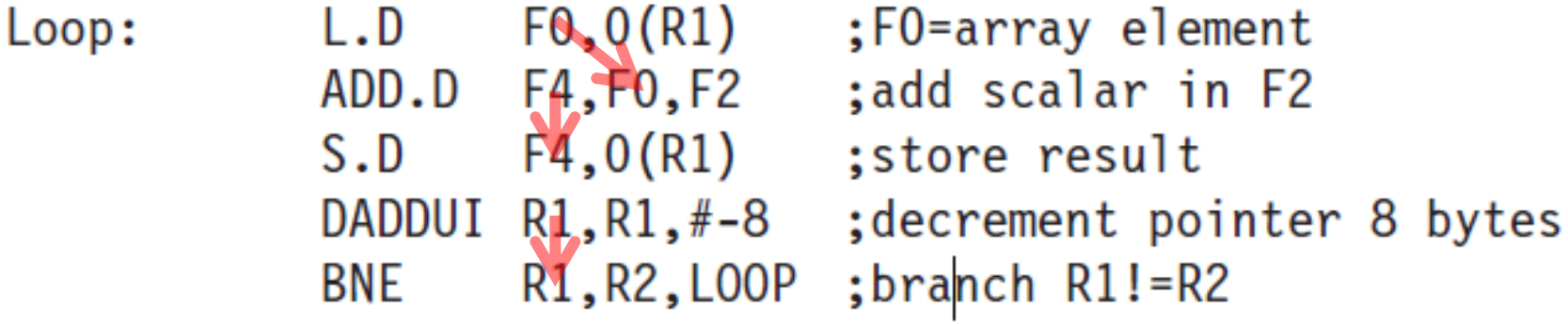
```
for (i=1; i<=1000; i=i+1)
        x[i] = x[i] + y[i];
```

No overlap opportunity within iteration, but every iteration can overlap with any other.

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.

Loop increments a vector in memory by a scalar in F2, starting at 0(R1), with last index at 8(R2)).



There are data dependences of floating-point and integer data.

Data dependent instructions cannot be executed simultaneously or completely overlap.

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: 100(R4) and 20(R6).
- Effective address of a load or store may change from one execution to another so that 100(R4) and 20(R6) may be different.

Compiler Techniques for Exposing ILP

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

To **avoid stall**, dependent instruction is **separated** from source by a distance (clock cycles) equals to **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 replicates the loop body multiple times.

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

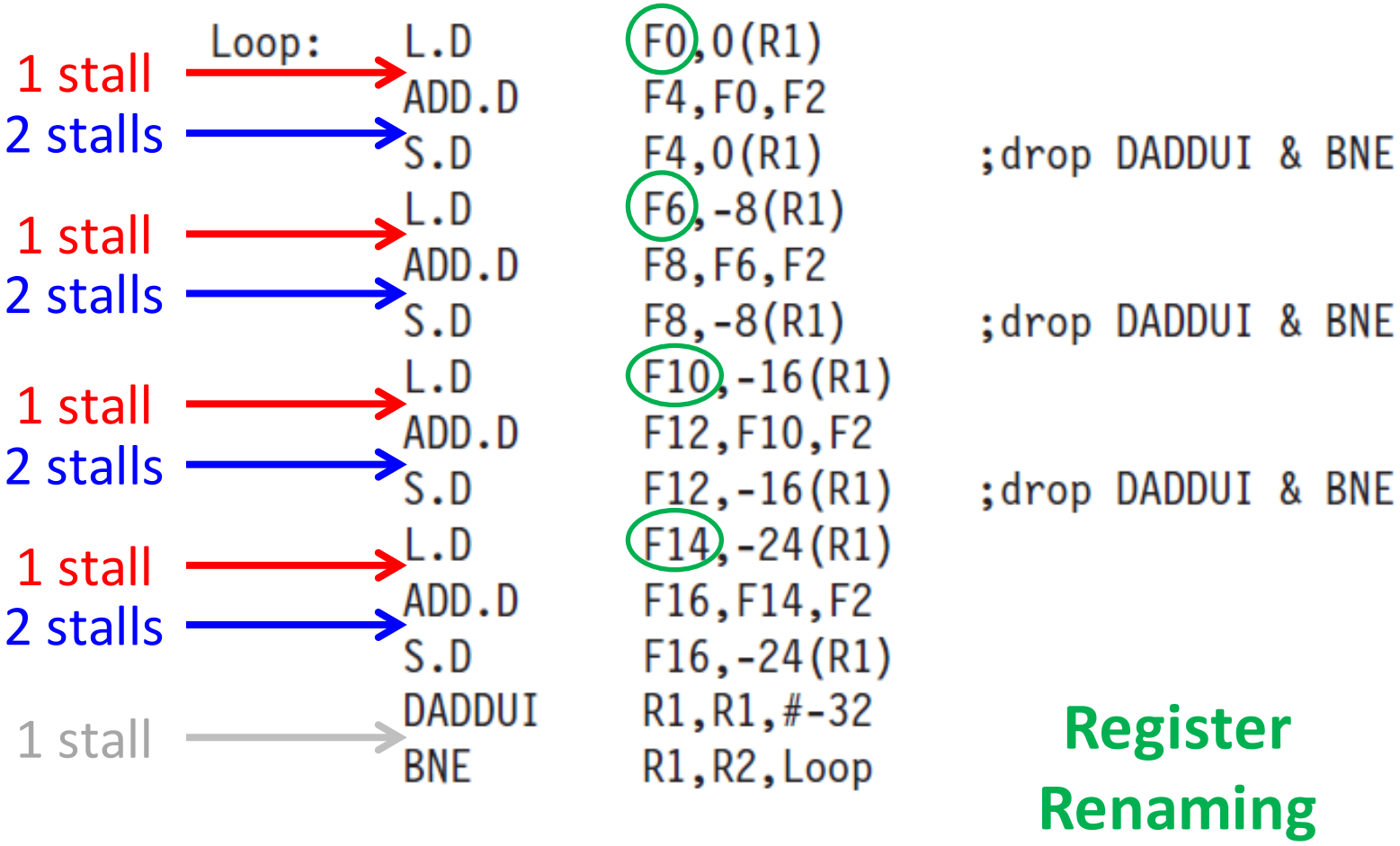
Instruction replication is insufficient. Different registers for each replication are required.

Required to increase the number of registers.

Useful work in the array is $3/7$ of cycles (load, add, and store). Other 4 are overhead.

Amortize overhead over more operations.

Unrolled code (not rescheduled)



Stalls are still there.

Runs 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 before unrolling or scheduling and 7 when scheduled but not unrolled.

Homework: Is it a hazard?

Problem: Number of iterations n may be unknown. Unrolling copies body k times, but $n \,\%\, k \neq 0$.

Homework: study the solution below.

Two consecutive loops are generated.

The first executes $n \,\%\, k$ with 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

Recall the performance losses occurred by control hazards, e.g., 3 stalls after conditional jumps.

Losses can be reduced by predicting how branches will behave.

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

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

Dynamic Branch Prediction

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

Note that BP may have been put there by another branch with same LSBs address bits!

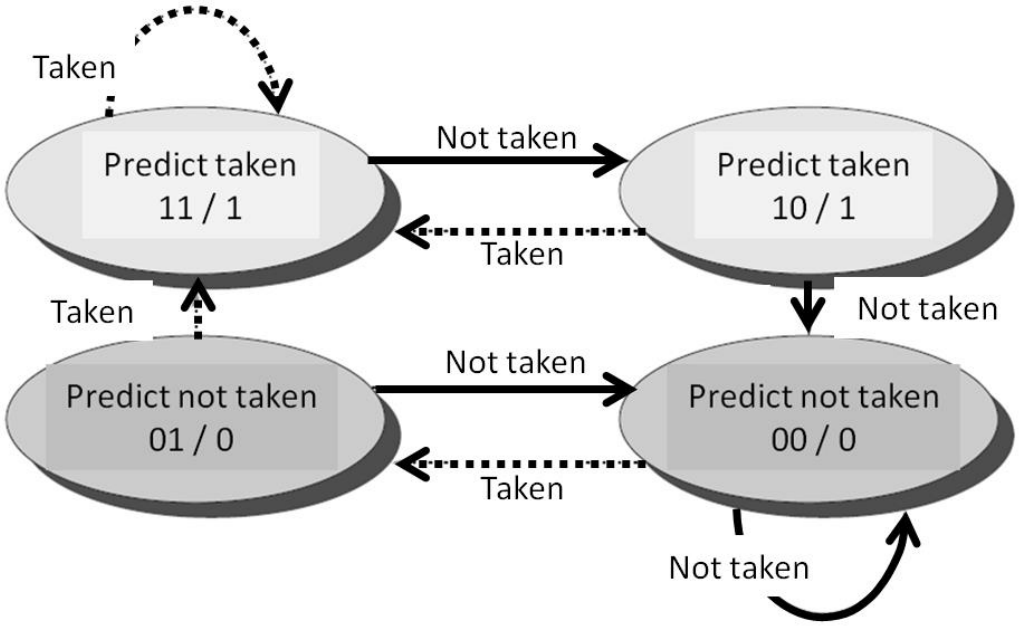
Useful when the branch delay (# stalls, e.g., $if\ \sin(x) < 0$ ...) is longer than the target PC address prediction.

IF 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 loop. Loop exit causes miss prediction. Re-entering the loop causes another miss prediction.

Solution:
saturation counter.



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

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. $\text{counter} \leq 2^{n-1} - 1$ yields not taken prediction. Otherwise, taken 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.

But branch result may depend on other branches.

Consider the code:

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

Compiler generates the typical MIPS code, assigning aa and bb to R1 and R2.

```
        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)
```

Behavior of b3 is correlated to that of b1 and b2.

Predictor using only the behavior of a single branch is blind of this behavior.

Correlating or **two-level predictors** add info about the most recent branches to decide the present branch.

An (m, n) BP uses m recent branches global history, (stored in m-bit shift register) to choose from 2^m n-bit branch predictors (per branch).

More accurate than 2-bit and requires simple HW.

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

Example: $(2,2)$ BP buffer with $64 = 2^6$ entries. 6-bit index is formed by 4 LSBs of branch address + 2 global bits of the two most recent branches outcome.

To compare **fairly** BPs performance, same total number of state bits are used.

Number of total bits of (m, n) predictor buffer storing branch addresses by their r LSBs is $2^{m+r} \times n$.

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

Example: How many bits in $(0,2)$ BP with $4K$ entries?

How many prediction entries (addresses) 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.

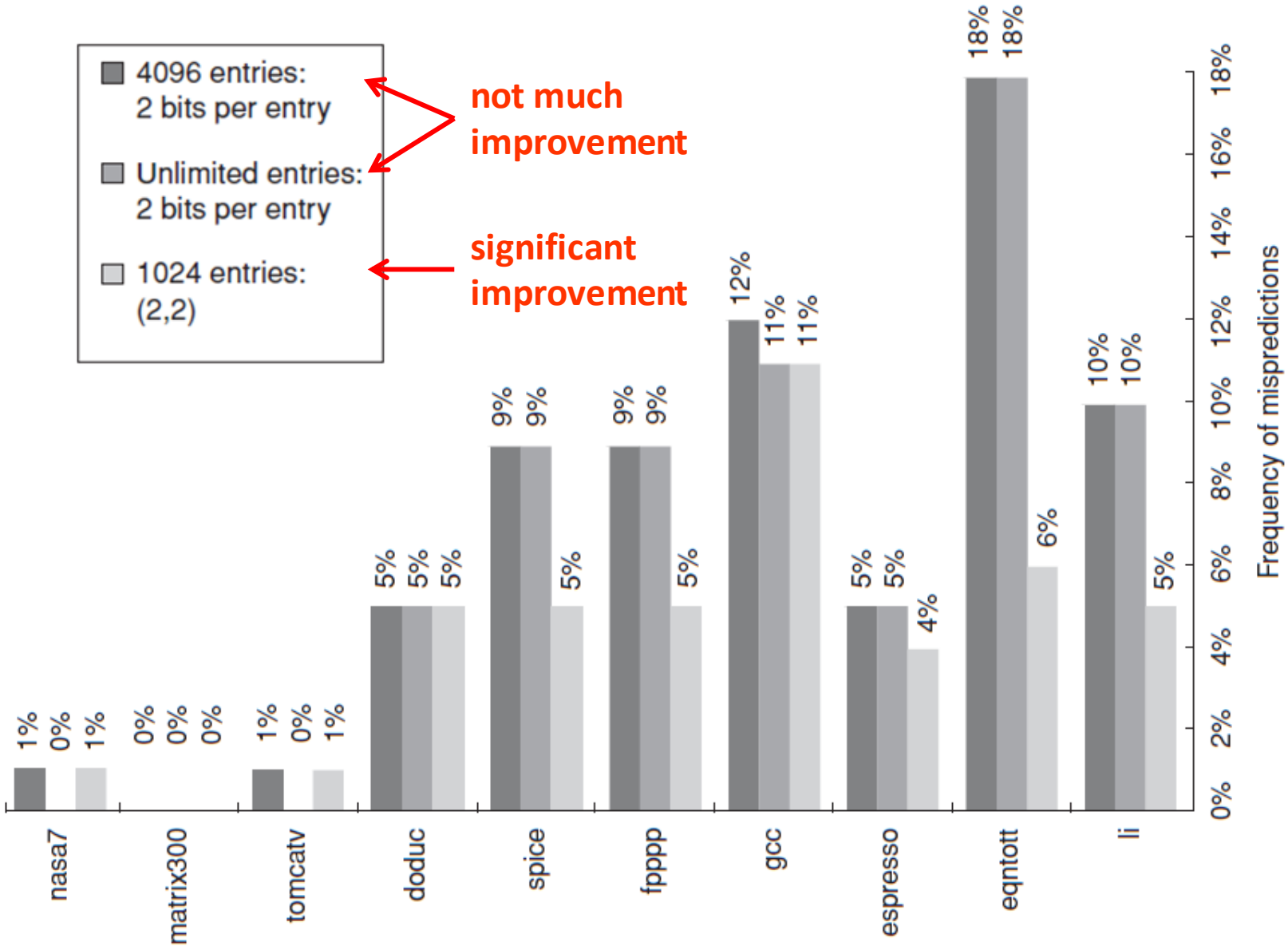
A $(2,2)$ BP having a total of 8K bits satisfies:

$$2^m \times n \times 2^r = 8\text{K}=2^{13}$$

$$2^2 \times 2 \times 2^r = 8\text{K}=2^{13}$$

entries selected by the branch address $= 8\text{K}$ bits.

$r = 10 \Rightarrow$ # of prediction entries (addresses) is 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%

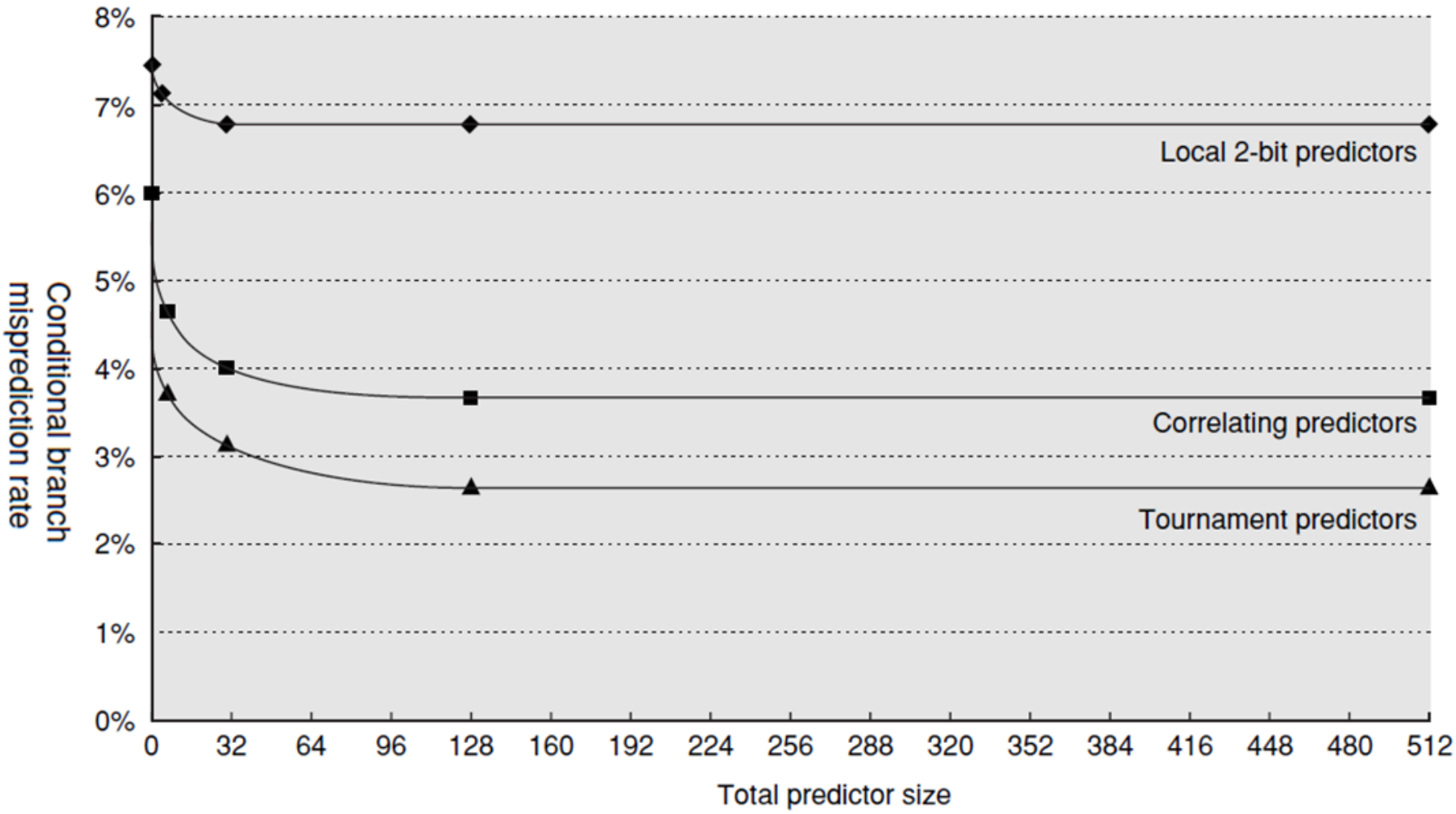
Tournament Predictors

Tournament predictors combine predictors based on global and local information.

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

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

Like simple 2-bit predictor, saturating counter requires two mispredictions before changing 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