DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

COMPUTER ARCHITECTURES

Branch Prediction

Gábor Horváth

BUTE Department of Networked Systems and Services
ghorvath@hit.bme.hu

Budapest,
04/29/2025

MŰEGYETEM 1782

CONTROL HAZARDS

- Control hazards
  - Caused by jump instructions
  - They break the sequential behavior of the instruction flow
- What is the problem?
  - E. g. in the case of conditional jumps the CPU must know
    - The outcome: if jump is taken or not
    - The target address: where to continue
  - The task of the IF stage: to load the instructions (from the instruction cache)
    - It has no time for
      - evaluation of the condition
      - address calculation
    - It has time for a hint
      - If it proves to be a good one: that is fine :-)
      - If not
        - Mistakenly started instructions must be invalidated
        - It learns from the mistake

THE TASK OF BRANCH PREDICTION

- Instructions that brake the sequential execution:
  - **Unconditional jump:**
    - Direct: `JUMP -28`
    - Indirect: `JUMP R1`
    - `-28` or `R1`: target address
  - **Conditional jump:**
    - Direct: `JUMP -28 IF R2>0`
    - Indirect: `JUMP R1 IF R2>0`
    - `R2>0`: jump condition
    - `-28` or `R1`: target address
  - **Subroutine call:**
    - Direct: `CALL -28`
    - Indirect: `CALL R1`
    - Return: `RET`, target address is from the stack
- The task of branch prediction is:
  - Predict the outcome, if there is a branch condition
  - Predict the target address

IMPORTANCE OF BRANCH PREDICTION

- What is the gain of a good prediction?
  - Instructions can be fetched without stopping
- What is the loss of a wrong prediction?
  - The later the error turns out, the more time is wasted
  - The longer the pipeline, the higher the loss is!

IMPORTANCE OF BRANCH PREDICTION

- Motivating example (Intel Core i7 Nehalem):
  - 4-way superscalar CPU
  - 17 clock cycles from IF to have branch condition evaluated and target address computed
  - Suppose that
    - every 4th instruction is a conditional jump
    - the hit rate of the branch predictor is 67%
    - there are no other hazards
  - 4-way superscalar means that throughput is 4 instructions/cycle
    → in ideal case **0.25 cycles/instruction** (on average)
  - 25% of instructions is a conditional jump, which wastes 17 cycles with the probability of 33%, thus instruction execution will need:
    → 0.25 + 0.25*0.33*17 = **1.65 cycles/instruction** (on average)
  - Perfect prediction would result in a 1.65/0.25=**6.6 times faster CPU!**

IMPORTANCE OF BRANCH PREDICTION

- Cycles lost due to a wrong prediction (=misprediction penalty)

CPU	Number of lost cycles
Intel Pentium I MMX	4-5
Intel Pentium 4	45 (on average!)
Intel Core2	15
Intel Core i7 Skylake	16.5
Intel Atom	13
AMD K8 és K10	12
AMD Ryzen	19
Via Nano	16
ARM Cortex A53	7
ARM Cortex A72	15

Prediction of the outcome of a branch

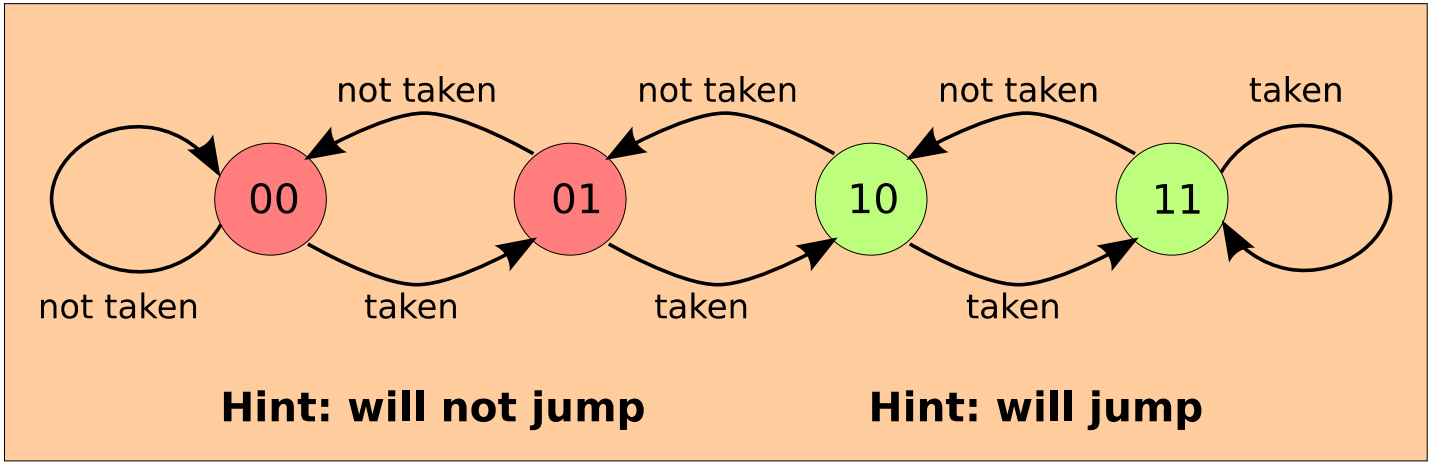
THE TASK TO SOLVE

- Known: the outcomes of the conditional jump in the past
- Example task: What will be the next outcome?
  - **1111111?**
  - **1111110110111011111?**
  - **110011001100 11?**
  - **11111111110000000000?**

- Requirements:
  - The predictor should be *fast*
  - The predictor should be *simple*
  - The predictor should have a *high success rate*

SIMPLE FINITE STATE MACHINE BASED SOLUTION

- A Finite State Machine (FSM) is assigned to every jump instruction
- The FSM records, how likely the instruction jumps
- If likelihood is >50%, jump is predicted
- FSM is refreshed on the basis of the **actual** behavior of the instruction



SIMPLE FINITE STATE MACHINE BASED SOLUTION

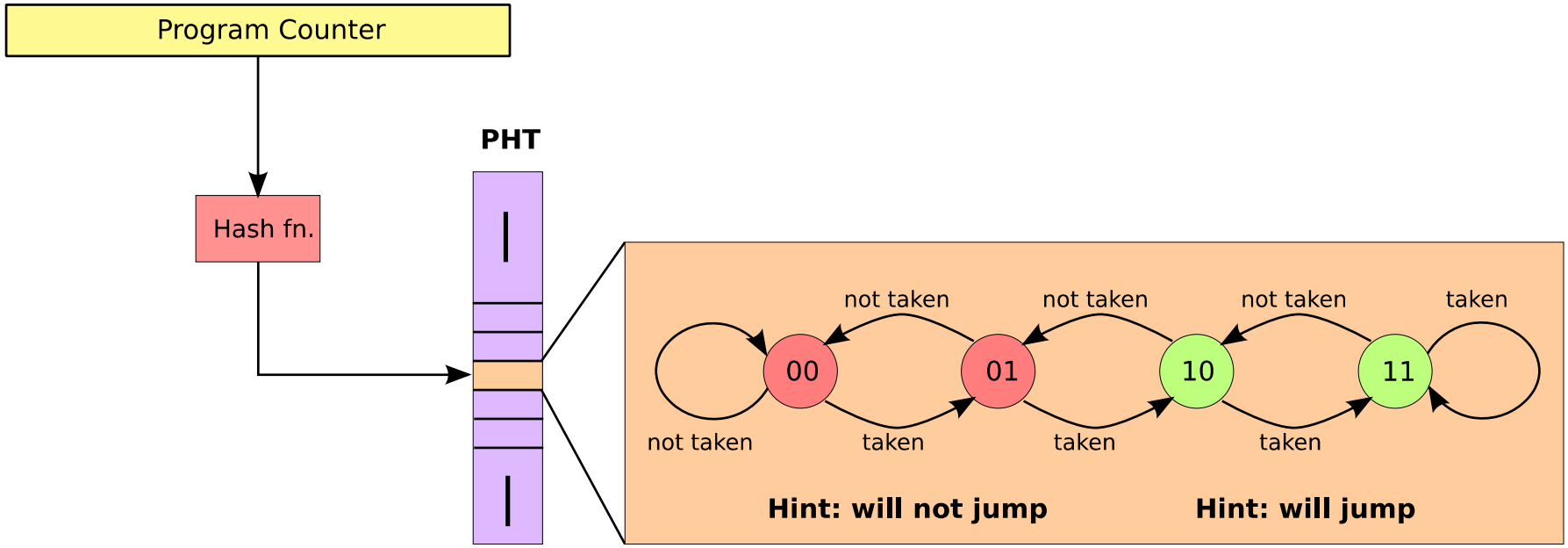
- Analysis:

```
for (i=0; i<m; i++) {
    for (j=0; j<n; j++) {
        ...
    }
}
```

- Let us examine the decisions of the internal for loop:
  - n*m decisions
  - using a single bit counter: 2*m wrong estimations
  - using a two bit counter: m wrong estimations

SIMPLE FINITE STATE MACHINE BASED SOLUTION

- Where shall we store the state of the FSM?
  - First idea: Let us use a cache like organization
    - 32 (or 64) bit tag, 2 bit data → it is not worth doing so!
  - In the block of the instruction cache (together with the instructions) → AMD
  - In a separate table: **PHT** (Pattern History Table) → Pentium 1



CONSIDERING CORRELATION

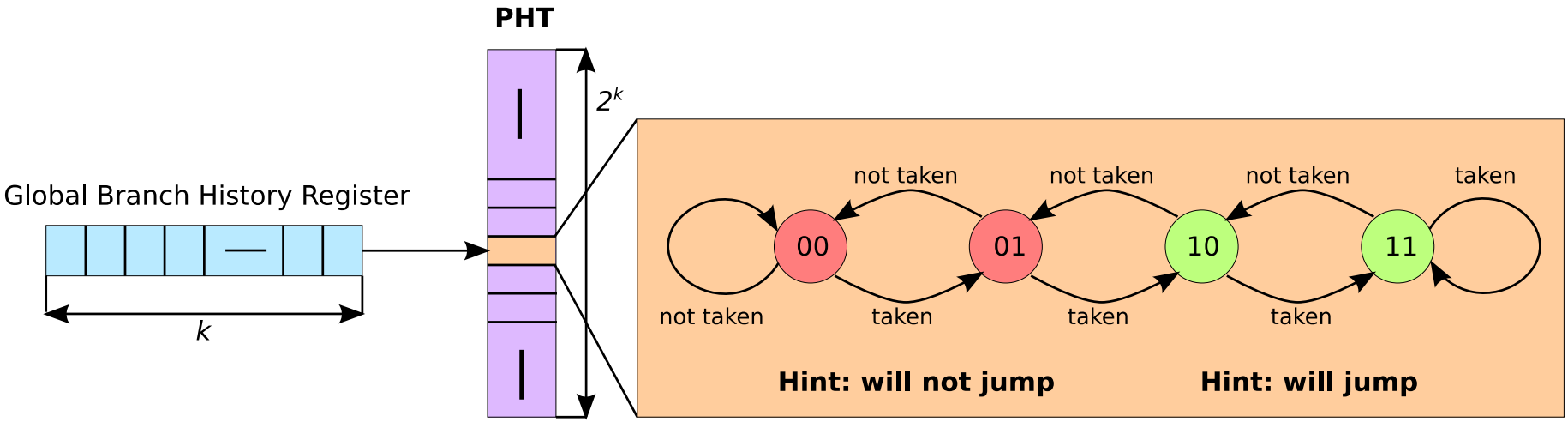
- The outcome of the conditional jump instructions often depends on the outcome of other conditional jump instructions
- Example:

```
if (a==2)
    a = 0;
if (b==2)
    b = 0;
if (a!=b) {
    ...
}
```

- If the first two conditions were true, then the third one will be false!
- It would be good to utilize such correlations!

CONSIDERING CORRELATION

- The trick: let us store the outcomes of the consecutive conditional instructions (0: not taken, 1: taken) in a shift register → GBHR
  Global Branch History Register
- The *actual* outcome of a conditional jump (0 or 1) enters from right
- A *k* bit GBHR stores the outcome of the last *k* conditional jumps
- GBHR is used to index the PHT

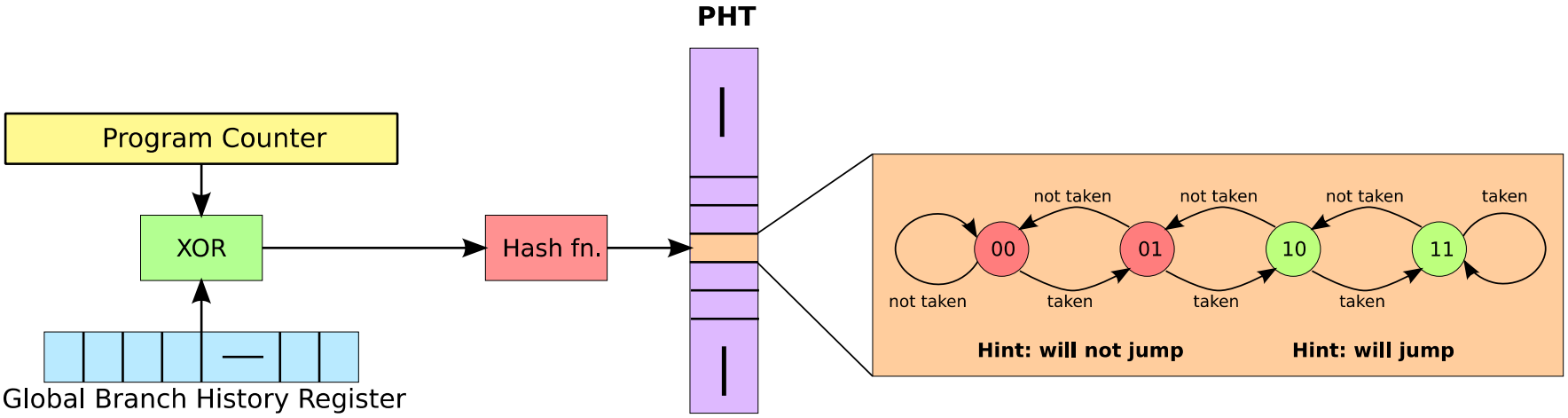


COMBINED PREDICTORS

- We have heard about
  - Simple finite state machine based solution
    - It makes an instruction local decision
  - Correlation based solution
    - It considers only global decision history
- Why don’t we combine the two?

THE GSHARE PREDICTOR

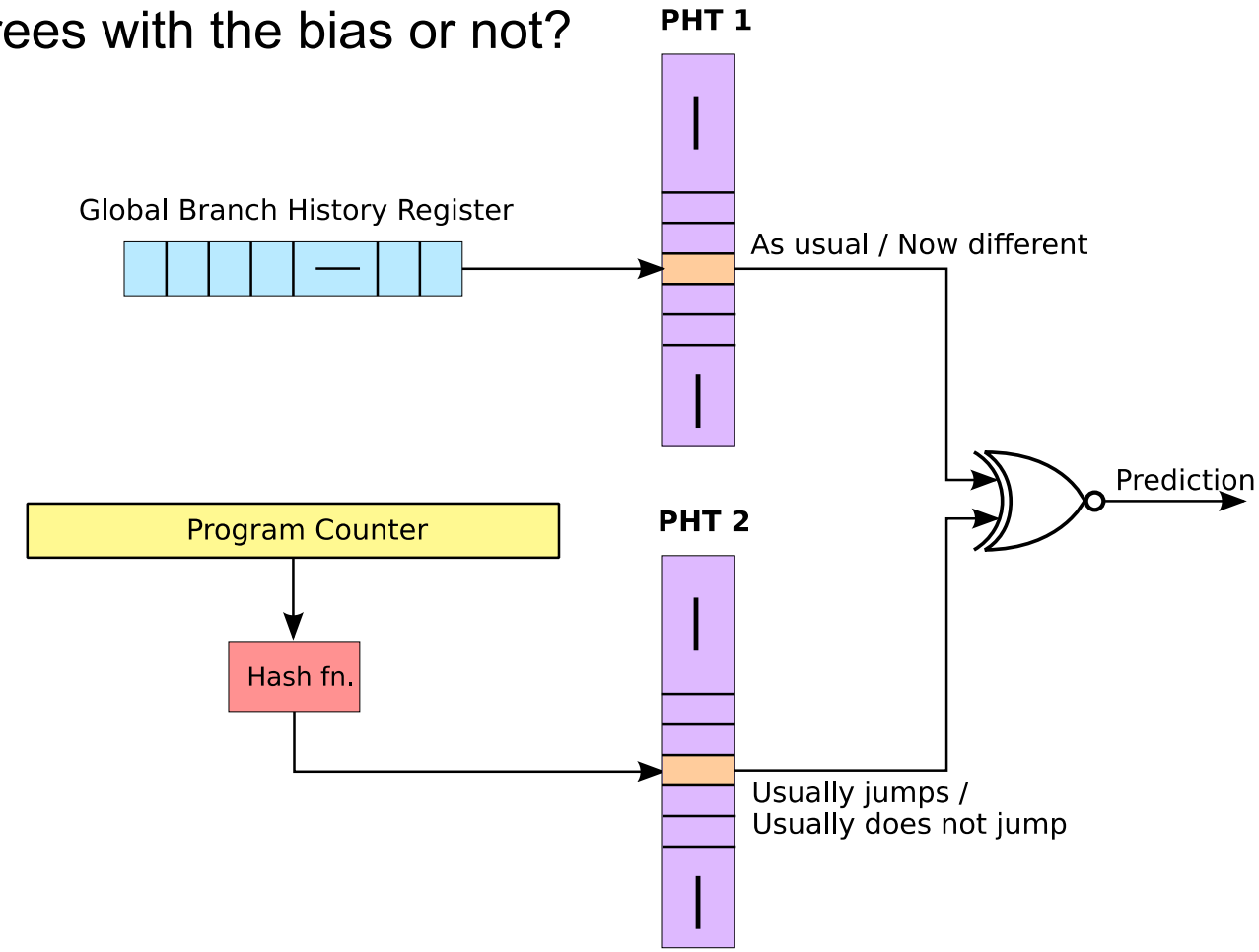
- It combines local and global pieces of information
  - Local decision: What PHT predicts for the given instruction?
  - Global decision: What PHT predicts on the basis of the global branch history?
  - Let us combine the two: PC XOR GBHR



- Very simple and surprisingly accurate!
- SPARC, POWER4, XBox 360, AMD Athlon, slightly modified version in ARM Cortex A8

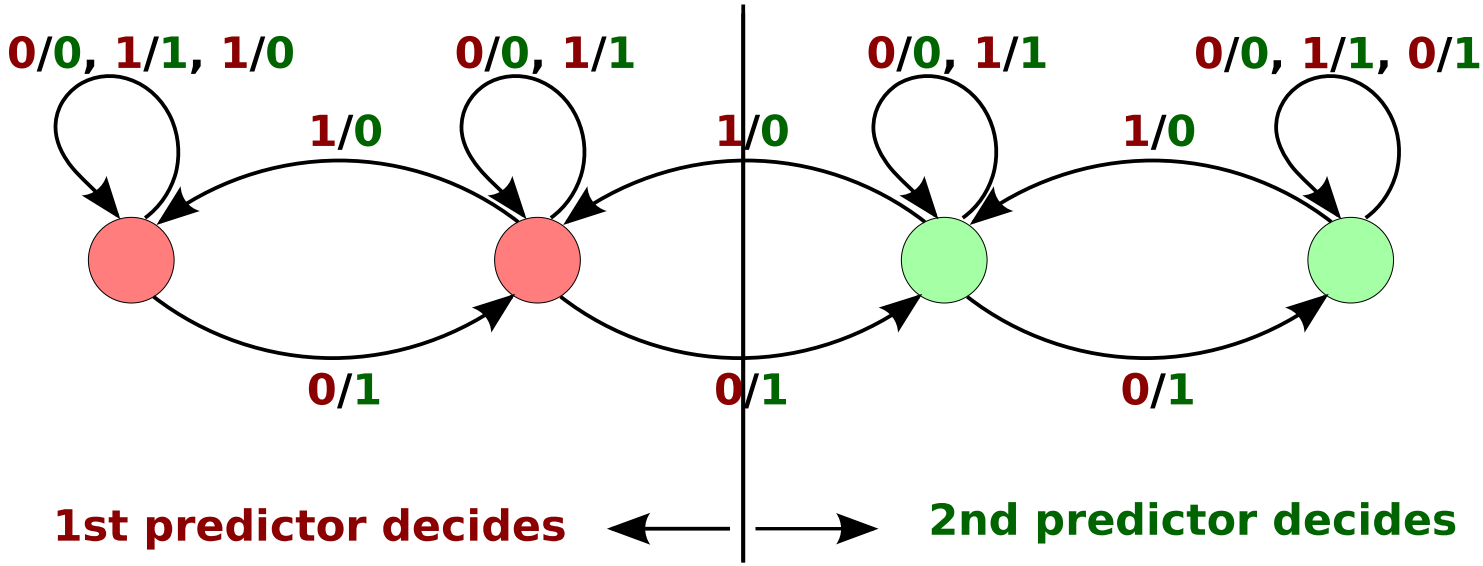
THE AGREE PREDICTOR

- Local procedure: the jumping “habits” of the given conditional jump (bias)
- Global procedure: this conditional jump now behaves as usual or not?
  - Agrees with the bias or not?



"TOURNAMENT" PREDICTORS

- Two predictors operate at the same time: a local one and a global one
- Always one of them is used for the decision: the one that has made better decisions recently
- A finite state machine keeps track, which predictor has been better nowadays



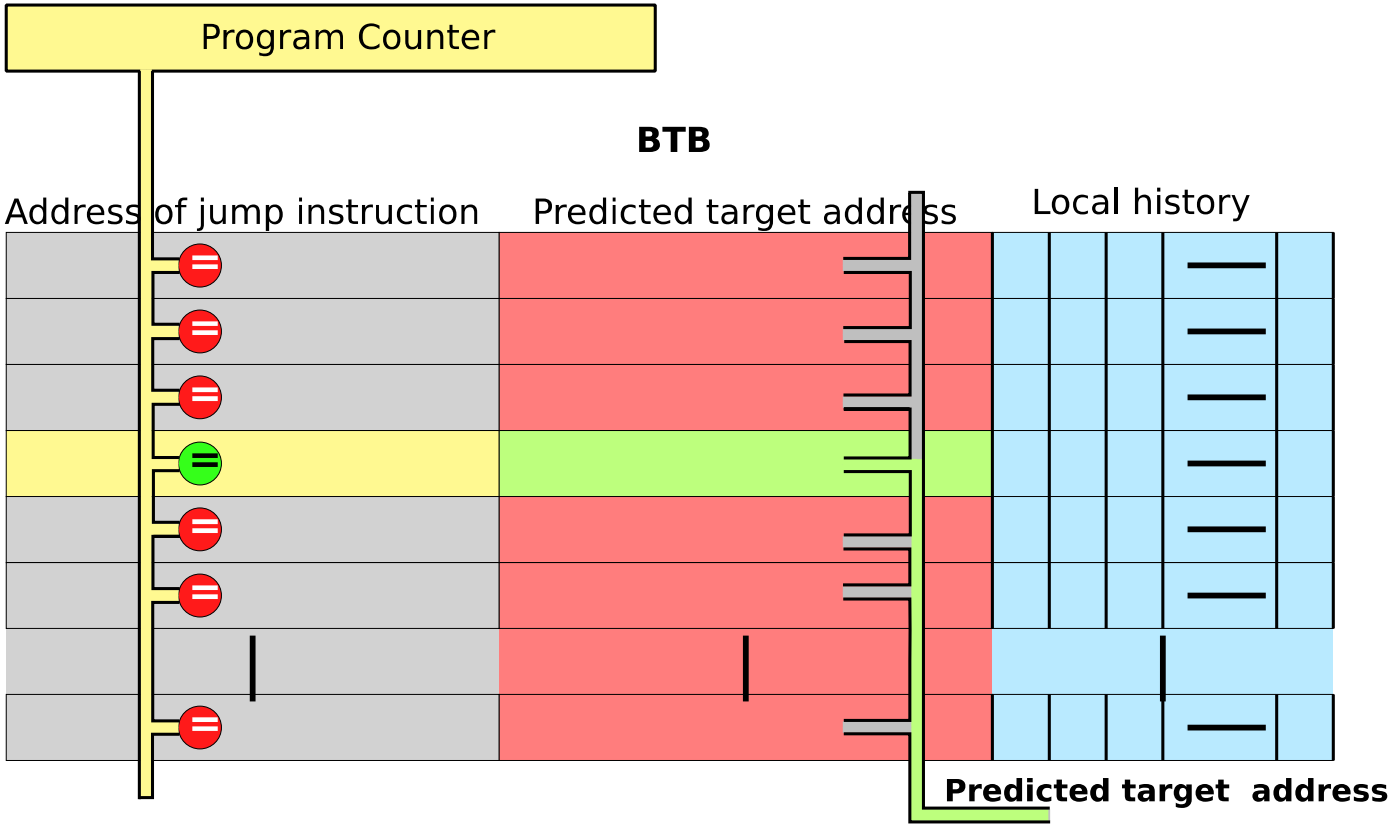
Branch target prediction

BRANCH TARGET PREDICTION

- From what address to fetch the next instruction?
  - It is very urgent for IF!
  - The question exists for both conditional and unconditional jumps.
- It is read from a special ***Branch Target Buffer*** (BTB)
  → It is easier/faster to read it, then to calculate it
  - The fields of the BTB:
    - tag: the address of the jump instruction
    - the expected target address
  - Organization:
    - A kind of cache organization, e. g. 4-way set associative
    - Content management: LRU
  - The stored value is corrected, if it proves to be wrong.

THE BRANCH TARGET BUFFER

- As it has an entry for every single (recently used) jump instructions, we can store further information in it (for conditional jumps):
  - e. g. Jumping habits, Local jump history, etc.
- For example:



RETURN ADDRESS PREDICTION

- The **RET** is a special jump instruction
  → return from subroutine call
  - The target address is in the slow memory (stack)
- An efficient data structure for prediction: **return stack**
  - It is in the CPU
  - High speed, low capacity
  - At the time of a subroutine call, the return address
    - Is put on the top of the real (slow) stack
    - … it is also put on the top of the return stack!
  - At the time of return
    - The CPU does not have to wait for the real (slow) stack
    - The return stack tells the target address within cycle time
  - It is efficient until it is becomes full
    - OK, if there is not too many nested function calls.

TYPICAL BRANCH TARGET BUFFER PARAMETERS

Processor	Num. BTB entries	BTB organization	Return stack
Intel Pentium I MMX	256	4-way set assoc.	no
Intel Pentium 4	4096	likely 8-way set assoc.	no
Intel Core2 (for cycles)	128	2-way set assoc.	16
Intel Core2 (indir. jump)	8192	4-way set assoc.	
Intel Core2 (other jump)	2048	4-way set assoc.	
Intel Atom	128	4-way set assoc.	8
AMD Steamroller	L1: 512, L2: 10240	L1: 4-way, L2: 5-way	24
AMD Ryzen	8/256/4096	?	31
Via Nano	4096	4-way set assoc.	very deep
ARM Cortex A8/A9	512	2-way set assoc.	8

POTENTIAL PROBLEMS

- How can we fool the branch target address predictor?
  → Let us jump in an unpredictable way!
  - E. g. traversing a heterogeneous collection
  - calling virtual functions in C++
  - An array containing pointers to functions
- A possible solution:
  - The problem is that BTB can store only one target address
  - Let it store more than one!
  - And the decision of the predictor, which address is to be used, is made as a function of the global branch history
  - E. g.: ARM Cortex A15, Intel Core Nehalem, AMD Ryzen, etc.

Branch prediction aware programming

ELIMINATING CONDITIONAL JUMP INSTRUCTIONS

- 1st example:

```
for (int i=0; i<N; i++)
    if (data[i] > 500)
        sum += data[i];
```

data[i]: random in [1, 1000]

- The conditional jump may be eliminated by a logical expression
  - `data[i]>500` ↔ `(data[i]-501) >= 0` (as they are integers)
  - Let us shift right: `data[i]-501>>31` (arithmetic shift, preserves sign!):
    - `000...0` (if >=0) or `111...1` (if <0)
  - Let us mask the addition operation with its complement
- Conditional jump is eliminated from the code:

```
for (int i=0; i<N; i++) {
    int t = (data[i]-501) >> 31;
    sum += ~t & data[i];
}
```

ELIMINATING CONDITIONAL JUMP INSTRUCTIONS

- 2nd example:

```
for (int i=0; i<N; i++)
    if (min<=data[i] && data[i]<=max)
        sum += data[i];
```

problem: *incomplete boolean evaluation* → conditional jump

- First step: unite the two conditions (somewhat nasty trick with binary complement)

`min<=data[i] && data[i]<=max` ↔ `(unsigned)(data[i]-min)<=max-min`

- Program code with only a single conditional jump:

```
for (int i=0; i<N; i++)
    if ((unsigned)(data[i]-min) <= max-min)
        sum += data[i];
```

- Second step: we use the trick shown in the 1st example:

```
for (int i=0; i<N; i++) {
    int t = (max-min-(unsigned)(data[i]-min)) >> 31;
    sum += ~t & data[i];
}
```

ELIMINATING CONDITIONAL JUMP INSTRUCTIONS

- 3rd example:

```
for (int i=0; i<N; i++)
   if (!((data[i]>='a' && data[i]<='z') || (data[i]>='A' && data[i]<='Z')))
        data[i] = ' ';
```

- Trick: we use an auxiliary array: **look-up table**, LUT
  - Specifies the substitution character for every single character
- LUT preparation + conditional jump free code:

```
for (int j=0; j<256; j++)
    if (!((j>='a' && j<='z') || (j>='A' && j<='Z')))
        LUT[j] = ' ';
    else
        LUT[j] = j;
for (int i=0; i<N; i++)
    data[i] = LUT[data[i]];
```

- LUT preparation time is constant, negligible if N is large enough

ELIMINATING CONDITIONAL JUMP INSTRUCTIONS

- Measurements with different architecture CPUs
  - N=2048*1024
- Results:

	i7-2600	Pentium 4	Rasp. Pi	RK3188
1st example, original	7,583 ms	14,122 ms	59,202 ms	11,296 ms
1st ex., cond. jump elimin.	1,297 ms	4,251 ms	58,410 ms	8,628 ms
2nd example, original	8,211 ms	19,6 ms	73,267 ms	21,496 ms
2nd ex. 1 cond. jump elim.	7,942 ms	14,295 ms	61,578 ms	13,347 ms
2nd ex. both c. jumps elim.	1,203 ms	4,252 ms	58,328 ms	8,268 ms
3rd example, original	6,533 ms	10,377 ms	48,532 ms	21,397 ms
3rd ex. with LUT	1,151 ms	3,641 ms	37,896 ms	18,240 ms

MAKING BRANCHES MORE PREDICTABLE

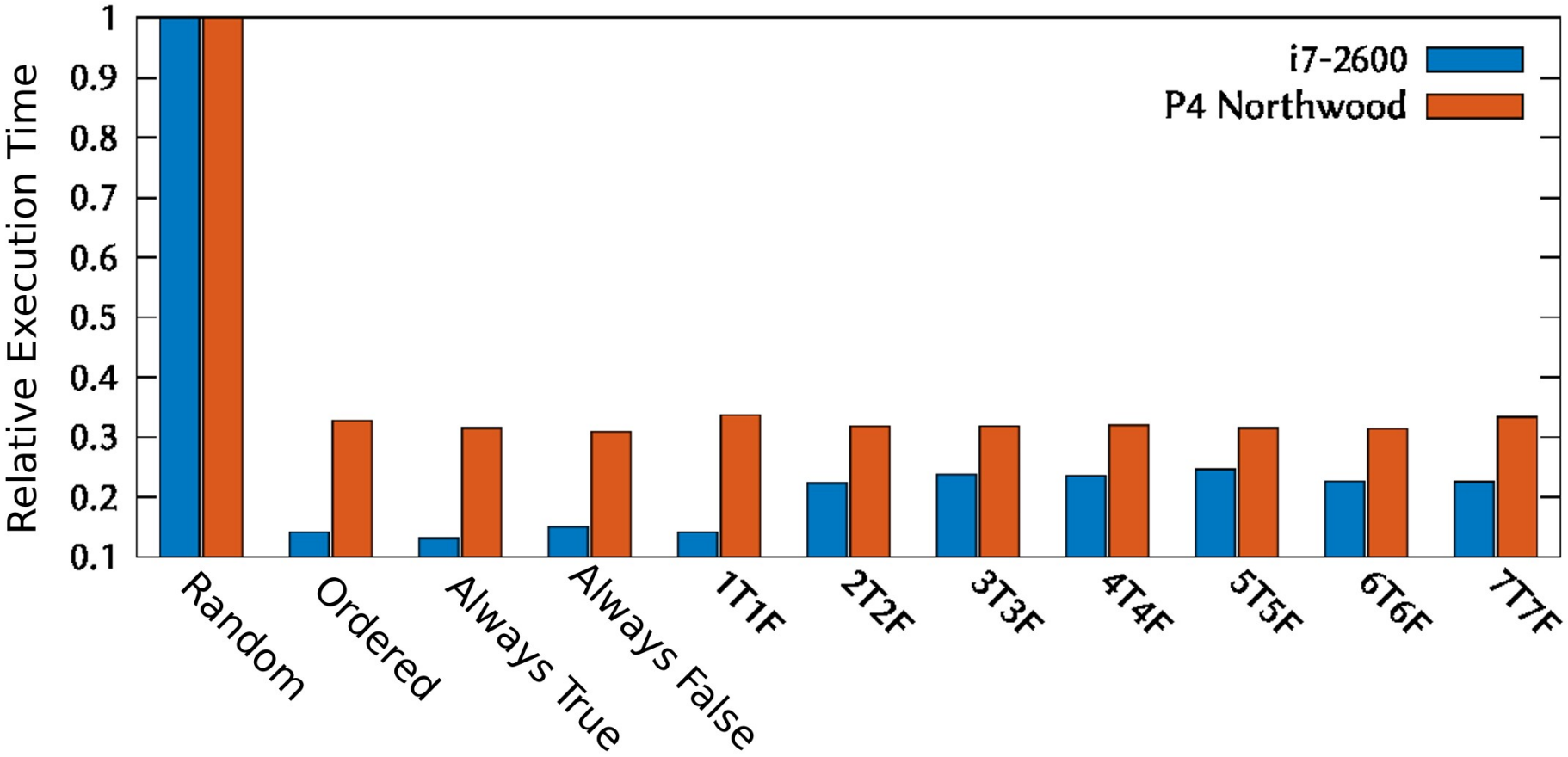
- Recall the 1st example:

```
for (int i=0; i<N; i++)
    if (data[i] > 500)
        sum += data[i];
```

- The elements are random in [1, 1000]
  → the outcome is unpredictable
- Let us sort the elements in increasing order!
- Measurements:
  - Unordered array
  - Ordered array
  - An array with *all* elements > 500
  - An array with *no* elements > 500
  - An array with a specified pattern (T: true, F: false)

MAKING BRANCHES MORE PREDICTABLE

- Measurement results:



MAKING BRANCH TARGET ADDRESS MORE PREDICTABLE

- Example: heterogeneous collection with 16 derived classes (B1..B16)

```
class A {
public:
    virtual int value ()=0;
    virtual int type ()
const =0;
    virtual ~A() {}
};
```

```
class B1 : public A {
    int v;
public:
    B1 () : v(rand()) {}
    int value () { return ++v; }
    int type () const { return 1; }
    ~B1 () {}
};
```

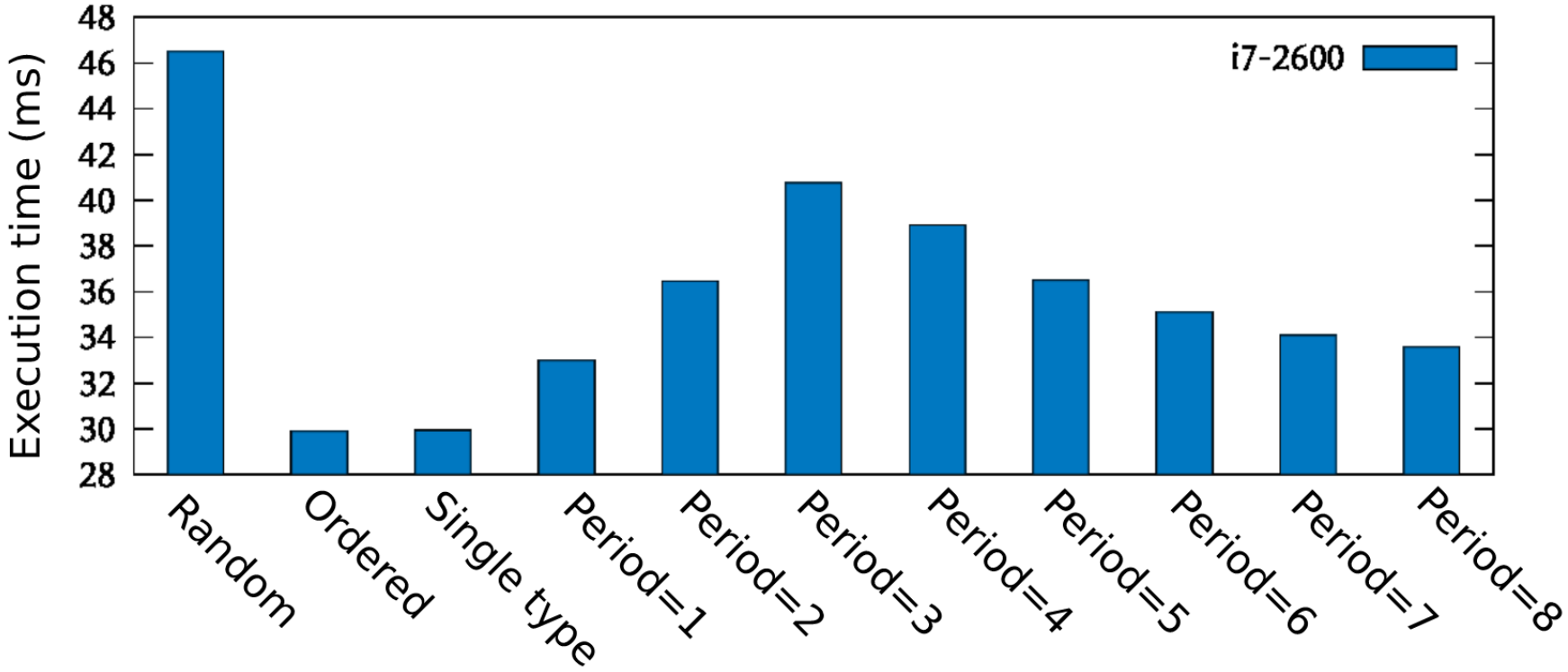
- Traversal:

```
sum = 0;
for (i=0; i<siz; i++)
    sum += data[i]->value();
```

- Problem:
  - Virtual function call is an **indirect** jump
    → Branch target address is unpredictable!
- Solution:
  - Let us sort the array by type!

MAKING BRANCH TARGET ADDRESS MORE PREDICTABLE

- Measurement results:



- Period=1: B1,B2,B3,…,B16,B1,B2,B3,….B16,B1,B2,B3,…
- ...
- Period=4: B1,B1,B1,B1,B2,B2,B2,B2,B3,B3,B3,B3,B4,B4,...

DEPARTMENT OF
NETWORKED SYSTEMS
AND SERVICES