分岐予測

文献

1. A PPM-like, tag-based branch predictor (2005)
   https://www.jilp.org/vol7/v7paper10.pdf
   (単体で読める)

2. A case for (partially) TAgged GEometric history length branch prediction (2006)
   https://www.jilp.org/vol8/v8paper1.pdf
   (1を先に読まないとわからない)

3. AMDがZen 2で採用した現在最強の分岐予測「TAGE」
   https://pc.watch.impress.co.jp/docs/column/kaigai/1192296.html

4. AMD Zen 2 CPUコアの物理的な姿が明らかに
   https://pc.watch.impress.co.jp/docs/column/kaigai/1237830.html

おさらい

用語

- イン・オーダー、アウト・オブ・オーダー実行
  - リザベーションステーション
- レジスタ・リネーミング
  - リオーダーバッファ
- 投機的実行(Speculative execution)

アウト・オブ・オーダー命令実行

- 命令の並びと実行開始・終了の順番が一致しない
- オペランドのデータ依存関係を解決して並列化
- パイプラインのフラッシュ(巻き戻し)は行わない

リザベーションステーション

- アウト・オブ・オーダー実行を実現するアルゴリズムの名前

レジスタ・リネーミング

- 命令ごとのデータ依存関係を減らすため、レジスタの割り付けをやり直すことで並列度を上げる技術
- アウト・オブ・オーダー実行に必須ではない

リオーダーバッファ

- レジスタ・リネーミングを実現するアルゴリズムの名前

投機的実行

- 条件付き分岐(beqとか)を発見したとき、分岐する/しないが確定してないけど予測して試しに実行してみる
- 分岐する/しないが確定したとき、予測が間違ってたらパイプラインをフラッシュ(巻き戻し)して分岐先からやり直す
- パイプラインが深い現代のCPUでIPCを劇的に改善できる(予測精度が95%でも理想値から30%も性能低下する)

なぜ必要?

- アウト・オブ・オーダー実行

  ➡ ハードブロックの稼働率を上げIPCを向上する

- 投機的実行

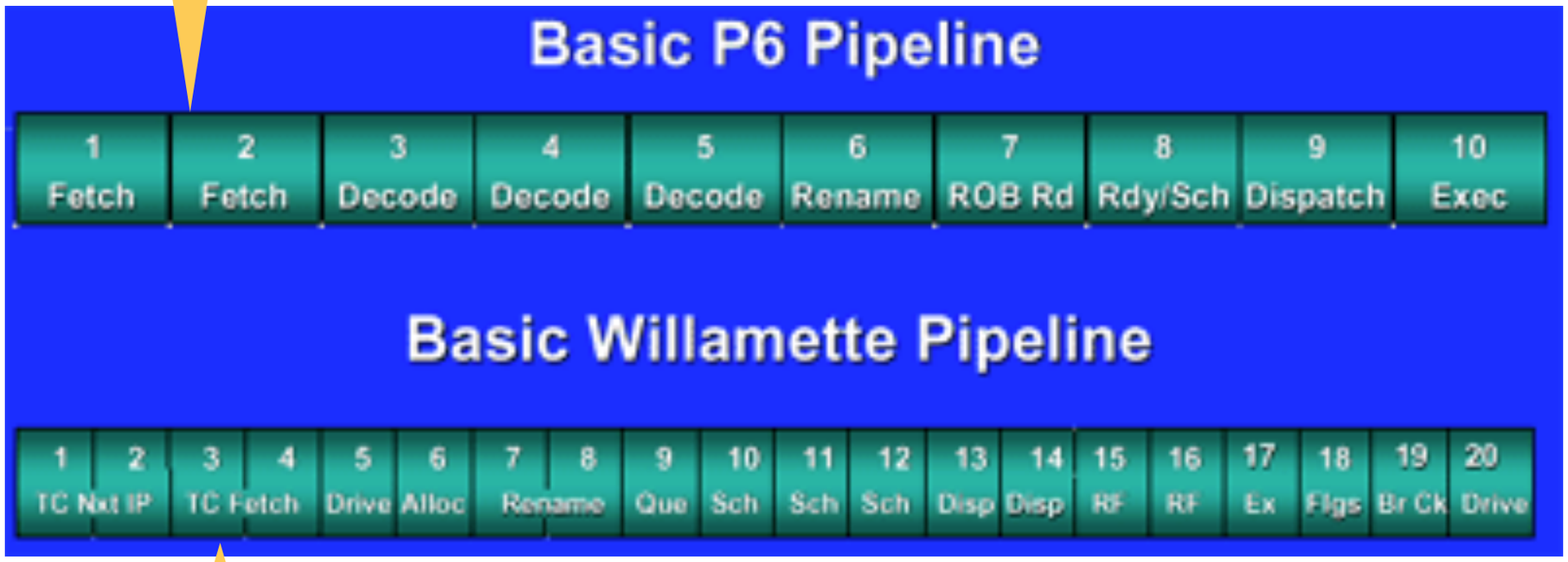
  ➡ 分岐でのストールを減らしIPCを向上する

どちらもシングルスレッド性能を改善する技術

シングルスレッド性能を上げるには

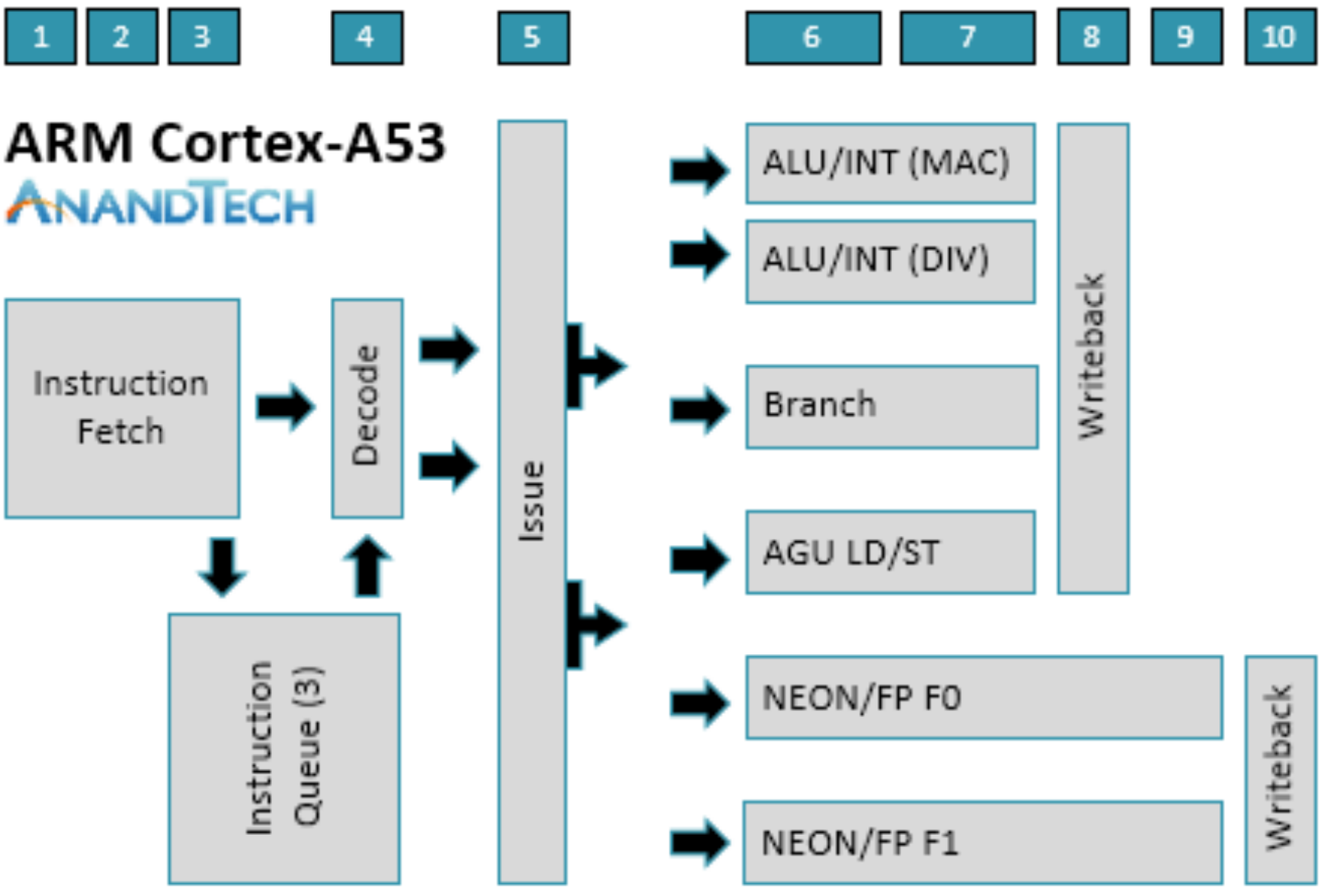
- 休まずパイプラインに命令を供給する

~PENTIUM 3

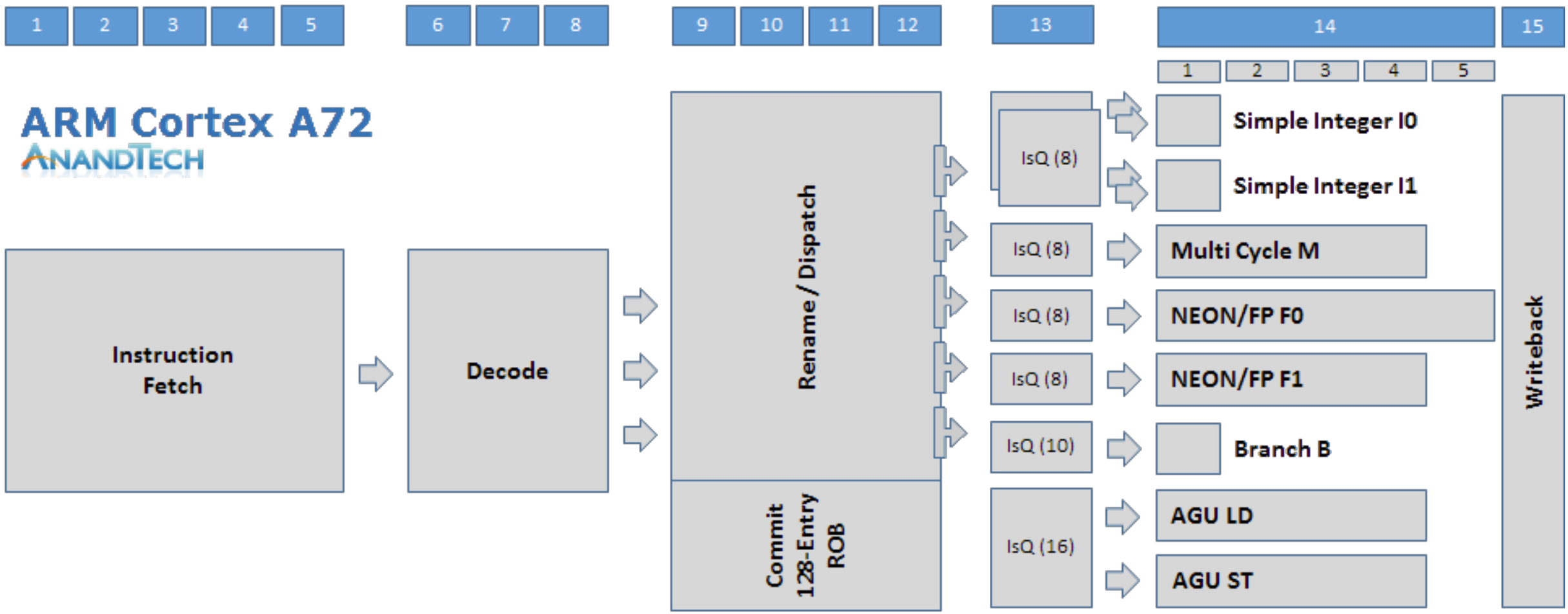


PENTIUM 4 第一世代

https://www.anandtech.com/show/604/3



https://www.anandtech.com/show/11441/dynamiq-and-arms-new-cpus-cortex-a75-a55/4



https://www.anandtech.com/show/10347/arm-cortex-a73-artemis-unveiled/2

単純な予測方法

必ず分岐する(しない)と予想する

- 分岐しないかもしれないが、するものとして続きを実行する
- コンパイラは分岐しないときのコストが高いことを知ってるので、条件付き分岐命令で分岐するコードを生成する
- バリエーションとしてランダム分岐予測がある(的中率50%)

分岐すると予測するほうが速い

```
    mov r0, #0
1:  add r0, r0, #1
    cmp r0, #10
    blt 1b
    …
```

分岐しないと予測するほうが速い

```
    mov r0, #0
1:  add r0, r0, #1
    cmp r0, #10
    bge 2f
    b   1b
2:  …
```

メモリをもつ分岐予測の基本的な考え方

- 条件付き分岐のほとんどはループ処理である

  ➡ 分岐する/しないは前回と同じであることが多い

- 分岐の履歴を記録すれば次回の動作を予測できる！

ローカル分岐予測

- 条件付き分岐命令が現れるアドレスと、分岐回数を表す飽和カウンタのテーブルを持つ
- 分岐したら+1, しなかったら-1
- MSB=1 → 今回も分岐するだろう...

タグ (アドレス)	分岐回数	次回予測
0000	01	分岐しない
0004	10	分岐する
...	...	
fffc	xx	

グローバル分岐予測

- グローバルな分岐の履歴を表すビットパターンと、分岐回数を表す飽和カウンタのテーブルを持つ

分岐履歴	分岐回数	次回予想
前回分岐しなかった 0000 前々回分岐しなかった	01	分岐しない
前回分岐した 0001 前々回分岐しなかった	10	分岐する
...	...	
ffff	xx	

2レベル分岐予測

- ローカル分岐予測とグローバル分岐予測の組み合わせ
- 縦にアドレス、横にグローバル分岐履歴のテーブル

	分岐履歴			
タグ	00	01	10	11
0000	00	10	00	11
0004	10	00	00	10
...	...	...	...	...
fffc	xx	xx	xx	xx

Cortex-A53の場合

Branch Target Instruction Cache

The IFU contains a single entry *Branch Target Instruction Cache* (BTIC). This stores up to two instruction cache fetches and enables the branch shadow of predicted taken branch instructions to be eliminated. The BTIC implementation is architecturally transparent, so it does not have to be flushed on a context switch.

Branch Target Address Cache

The IFU contains a 256-entry *Branch Target Address Cache* (BTAC) to predict the target address of indirect branches. The BTAC implementation is architecturally transparent, so it does not have to be flushed on a context switch.

Branch predictor

The branch predictor is a global type that uses branch history registers and a 3072-entry pattern history prediction table.

Return stack

The IFU includes an 8-entry return stack to accelerate returns from procedure calls. For each procedure call, the return address is pushed onto a hardware stack. When a procedure return is recognized, the address held in the return stack is popped, and the IFU uses it as the predicted return address. The return stack is architecturally transparent, so it does not have to be flushed on a context switch.

DDI0500
Cortex-A53 r0p4 TRM
2.1.1 Instruction Fetch Unit

TAGE

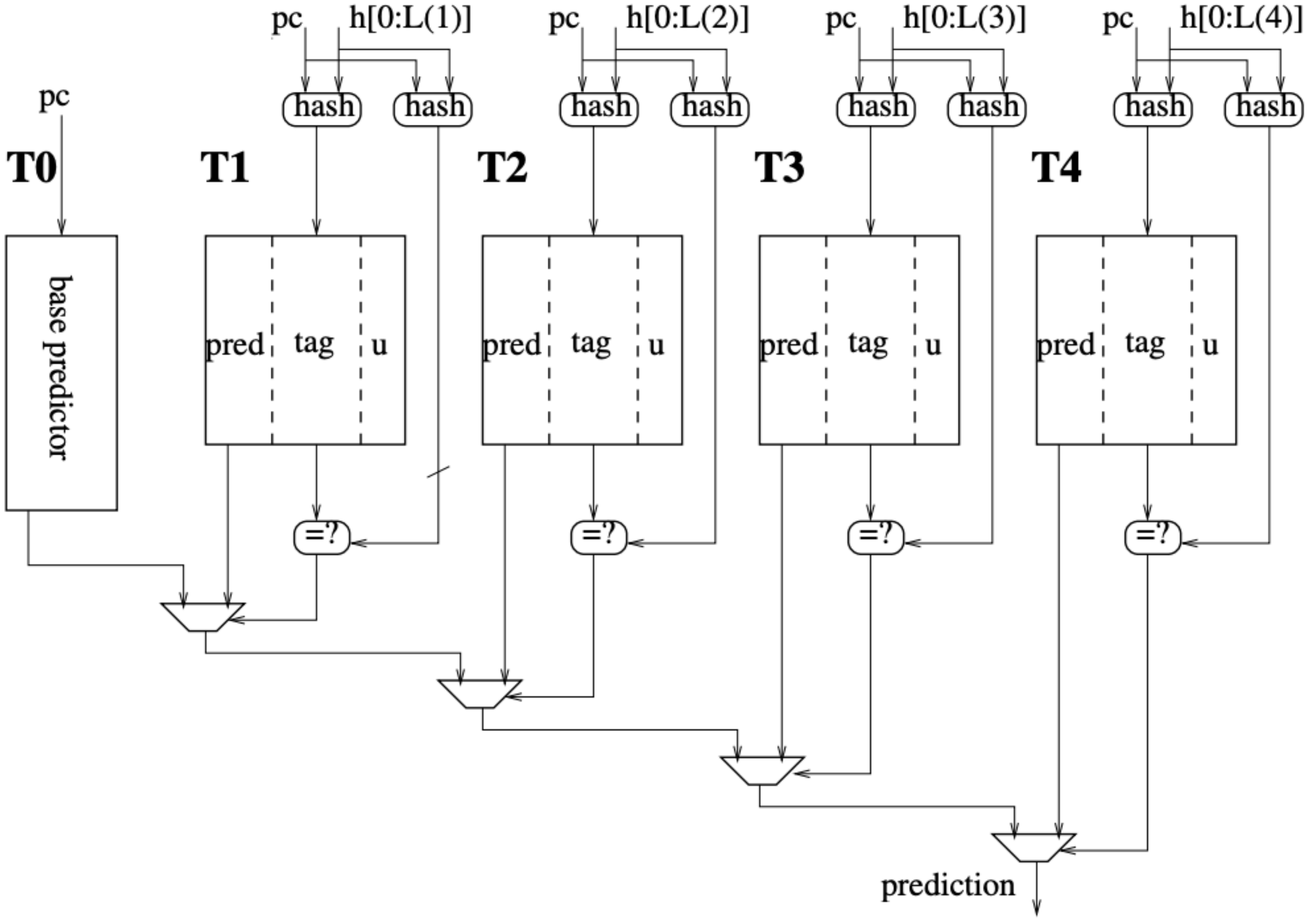


Figure 1: A 5-component TAGE predictor synopsis: a base predictor is backed with several tagged predictor components indexed with increasing history lengths

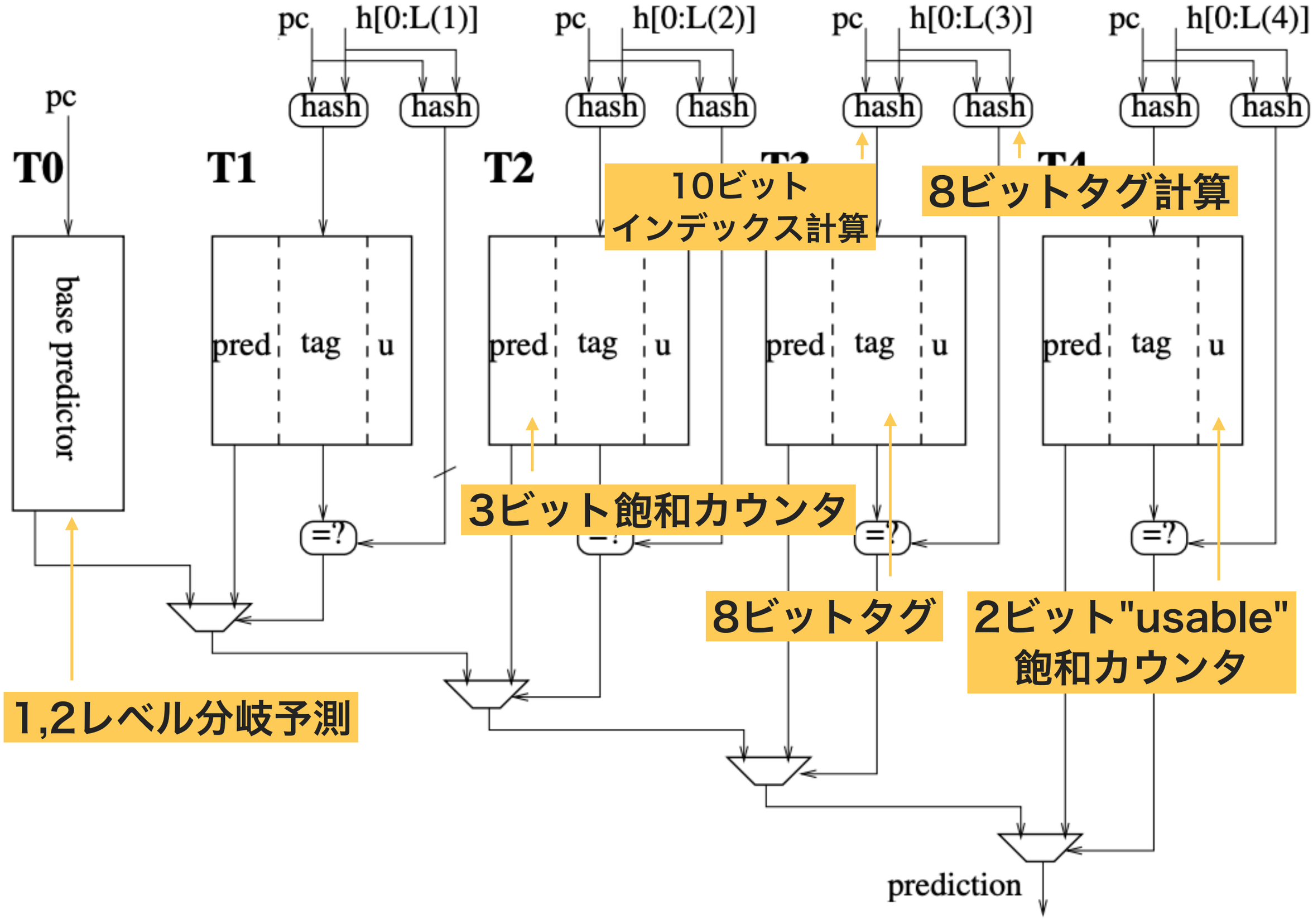


Figure 1: A 5-component TAGE predictor synopsis: a base predictor is backed with several tagged predictor components indexed with increasing history lengths

10 bit index	pred	8 bit tag	u
000000000	000	11111111	00
000000001	011	00000000	10
...	...	...	...
111111111	xxx	xxxxxxxx	x

疑似コード1

```
// iは0スタート
#define L(i) (i*10 - 1)                          // 文献1
#define L(i) (alpha**i * L(1))                   // 文献2
int pred_from_Ti(i, pc, h)
{
    index = calc_index(pc, h[0:L(i)]);
    tag   = calc_tag  (pc, h[0:L(i)]);
    if (GET_TAG(T[i][index]) != tag)
        return ERROR;                                    // エイリアス
    return ((GET_PRED(T[i][index]) >> 2) & 1    // 飽和カウンタMSBをとる
            ? true                                       // 分岐すると予想
            : false                                      // 分岐しないと予想
           );
}
```

疑似コード2

```
int pred(pc, h)
{
    for (i = 4; i >= 1; i--) {
        result = pred_from_Ti(i - 1, pc, h);
        if (!IS_ERROR(result))
            return result;
    }
    // fallback to base predictor
    return pred_base(pc, h);
}
```

疑似コード3

```
int update(i          // 予測に使ったテーブル
           pc, h,
           pred,      // 最終的な予測内容
           succeed) // 予測は成功したか?
{
    for (k = 0; k < i; k++) {
        result = pred_from_Ti(k, pc, h);
        if (!IS_ERROR(result) && result != pred)
           // 最終予想と異なる予想が存在した
           // 最終的に成功したなら+1, 失敗したなら-1
           update_usable_counter(i, pc, h, succeed ? 1 : -1);
    }
    update_pred_counter(i, pc, h, succeed ? 1 : -1);
    if (!succeed && i < 3) {
        // 長い履歴を使うテーブルに新しい行を追加
        // どのテーブルに追加するか(k)は各テーブルのusableカウンタの値によって決める
        k = calc_alloc_k(i, pc, h);
        alloc_row(k, pc, h);
    }
}
```

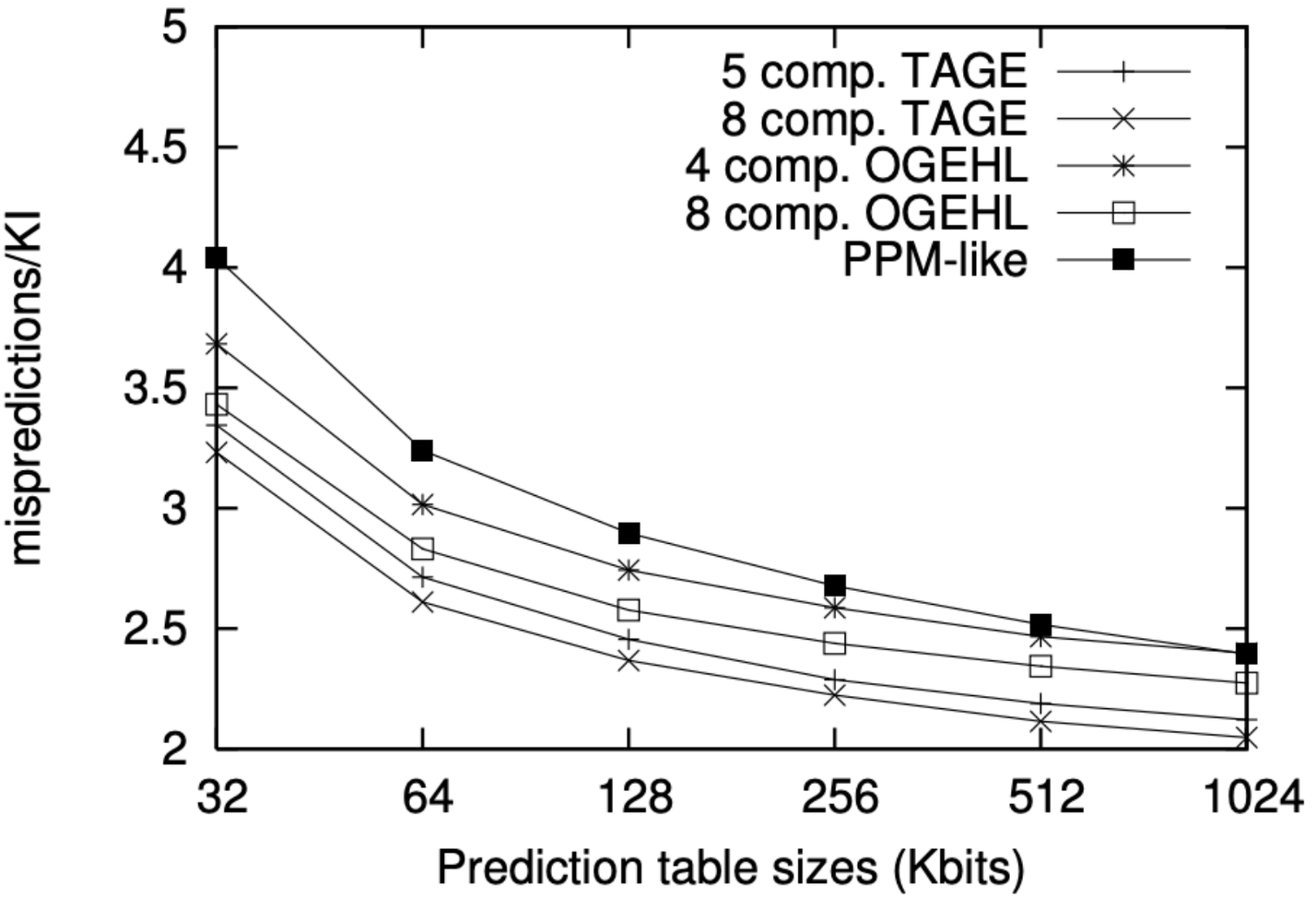


Figure 2: Conditional branch prediction accuracy on the TAGE predictor