PROJECT TITLE:

Adaptive Branch Predictor

ACA END_SEM PROJECT:

GROUP NO: 13

VARSHA SWARAJ (22CS02005)

VARRI NAVYA (22CS02010)

SUBMISSION: 11TH Nov 2025

Introduction

OBJECTIVE:

Here we have tried two different approaches and observed what works best for the adaptive case:

a) “To design, implement, and evaluate an Adaptive Gshare predictor in the SimpleScalar simulator and compare its performance with the baseline 2-level predictor.”
b) “To design, implement, and evaluate a Hybrid Branch Predictor combining bimodal and two-level adaptive prediction schemes in the SimpleScalar simulator, and to compare its performance with the baseline 2-level adaptive predictor”

Theoretical Background

1. **Two-Level Adaptive Prediction (2lev):**

   The 2lev predictor uses two levels of history to improve branch prediction. The Branch History Register (BHR) stores recent branch outcomes, and the Pattern History Table (PHT) uses this history to index 2-bit saturating counters that predict whether a branch will be taken or not. It adapts dynamically to program behavior.

2. **Gshare Predictor:**

   The Gshare predictor combines the program counter (PC) and global branch history using an XOR operation to index the prediction table. This reduces aliasing and improves accuracy. Each entry uses a 2-bit counter to track the branch tendency. In this project, two Gshare predictors with different history lengths are combined for better accuracy.

3. **Hybrid Branch Predictor:**

   A hybrid predictor combines a bimodal and a two-level predictor, using a meta (combining) table to choose which one to trust for each branch. Bimodal works well for simple, regular branches, 2-Level handles complex, correlated branches, Meta predictor learns which of the two is more accurate for each branch address.This combination reduces overall mispredictions and improves accuracy compared to using either predictor alone.

PART1: ADAPTIVE GSHARE BRANCH PREDICTOR (BY-VARSHA SWARAJ)

Simulator Setup

The project was implemented using the **SimpleScalar 3.0** simulator, a widely used toolset for studying microprocessor architecture and branch prediction mechanisms.

Benchmark -test-fmath

Design of Adaptive Gshare Predictor:

The Adaptive Gshare predictor combines two separate Gshare predictors one with a **short history** and another with a **long history** — to capture both local and global branch correlations. A **chooser table** (meta-predictor) determines which Gshare's prediction to trust for each branch.

In this design:

The **short Gshare** uses a smaller history length (e.g., 6 bits) for fast adaptation to recent behavior.

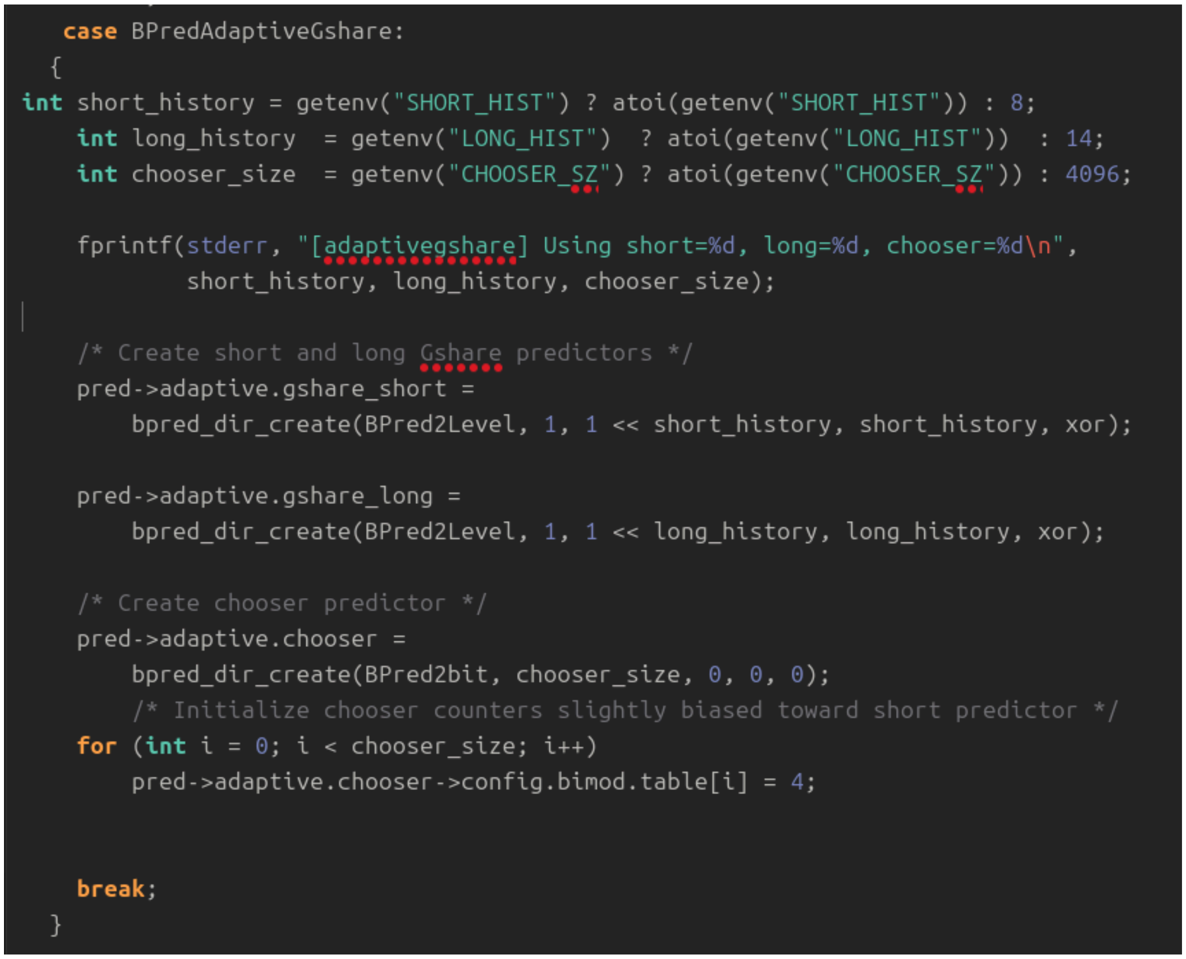
The **long Gshare** uses a larger history length (e.g., 20 bits) to learn long-term branch patterns.

The **chooser table** (typically 16K entries) uses 3-bit saturating counters to decide which predictor is more accurate in a given context.

Integration with SimpleScalar:

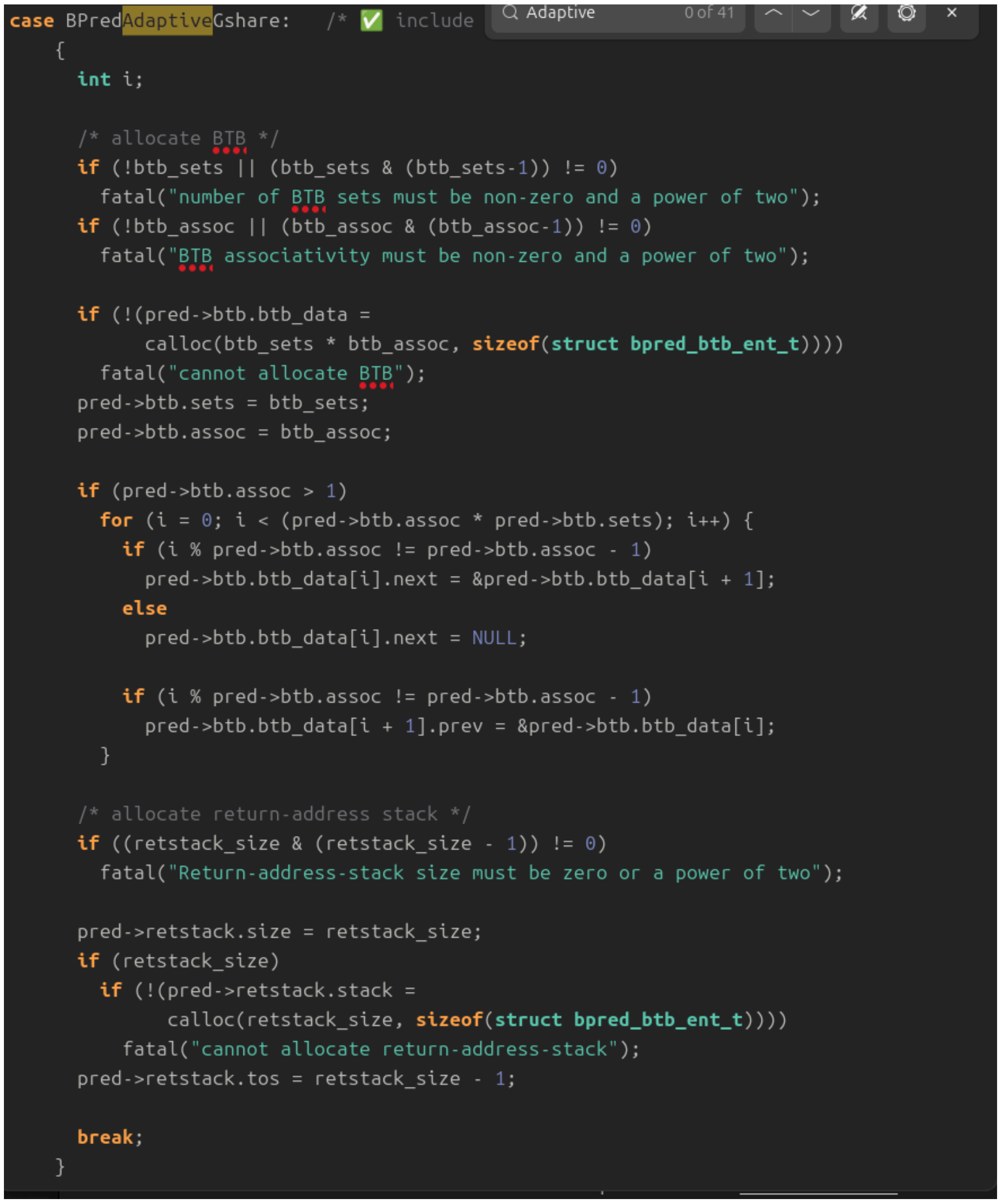
BPRED.C

A) In bpred_create():



```
    case BPredAdaptiveGshare:
  {
int short_history = getenv("SHORT_HIST") ? atoi(getenv("SHORT_HIST")) : 8;
    int long_history  = getenv("LONG_HIST")  ? atoi(getenv("LONG_HIST"))  : 14;
    int chooser_size  = getenv("CHOOSER_SZ") ? atoi(getenv("CHOOSER_SZ")) : 4096;

    fprintf(stderr, "[adaptivegshare] Using short=%d, long=%d, chooser=%d\n",
            short_history, long_history, chooser_size);

    /* Create short and long Gshare predictors */
    pred->adaptive.gshare_short =
        bpred_dir_create(BPred2Level, 1, 1 << short_history, short_history, xor);

    pred->adaptive.gshare_long =
        bpred_dir_create(BPred2Level, 1, 1 << long_history, long_history, xor);

    /* Create chooser predictor */
    pred->adaptive.chooser =
        bpred_dir_create(BPred2bit, chooser_size, 0, 0, 0);
        /* Initialize chooser counters slightly biased toward short predictor */
    for (int i = 0; i < chooser_size; i++)
        pred->adaptive.chooser->config.bimod.table[i] = 4;


    break;
  }
```

This block initializes the **Adaptive Gshare predictor** when selected.It reads history and chooser sizes from environment variables (or uses default values) and prints the chosen configuration. Two Gshare predictors are created — one with **short history** and one with **long history** — to capture different correlation lengths. A **chooser table** is also created using 3-bit saturating counters (initialized to 4) that decide which Gshare's prediction to use during execution.

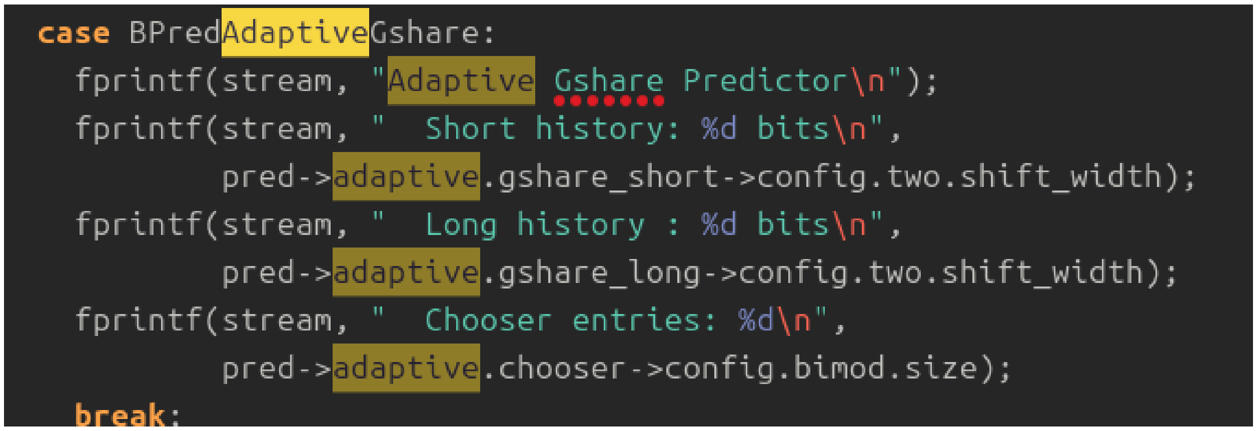


```
case BPredAdaptiveGshare:   /* ✅ include
    {
      int i;

      /* allocate BTB */
      if (!btb_sets || (btb_sets & (btb_sets-1)) != 0)
        fatal("number of BTB sets must be non-zero and a power of two");
      if (!btb_assoc || (btb_assoc & (btb_assoc-1)) != 0)
        fatal("BTB associativity must be non-zero and a power of two");

      if (!(pred->btb.btb_data =
            calloc(btb_sets * btb_assoc, sizeof(struct bpred_btb_ent_t))))
        fatal("cannot allocate BTB");
      pred->btb.sets = btb_sets;
      pred->btb.assoc = btb_assoc;

      if (pred->btb.assoc > 1)
        for (i = 0; i < (pred->btb.assoc * pred->btb.sets); i++) {
          if (i % pred->btb.assoc != pred->btb.assoc - 1)
            pred->btb.btb_data[i].next = &pred->btb.btb_data[i + 1];
          else
            pred->btb.btb_data[i].next = NULL;

          if (i % pred->btb.assoc != pred->btb.assoc - 1)
            pred->btb.btb_data[i + 1].prev = &pred->btb.btb_data[i];
        }

      /* allocate return-address stack */
      if ((retstack_size & (retstack_size - 1)) != 0)
        fatal("Return-address-stack size must be zero or a power of two");

      pred->retstack.size = retstack_size;
      if (retstack_size)
        if (!(pred->retstack.stack =
              calloc(retstack_size, sizeof(struct bpred_btb_ent_t))))
          fatal("cannot allocate return-address-stack");
      pred->retstack.tos = retstack_size - 1;

      break;
    }
```

This block sets up the **branch target buffer (BTB)** and **return-address stack (RAS)** for the Adaptive Gshare predictor.

It ensures the BTB's number of sets and associativity are valid powers of two, allocates memory for all BTB entries, and links them for LRU management.It then creates the RAS, which tracks return addresses for function calls, ensuring correct prediction of `return` instructions.Together, these structures allow the predictor to store and retrieve branch targets efficiently.

B) In bpred_config():



```
case BPredAdaptiveGshare:
  fprintf(stream, "Adaptive Gshare Predictor\n");
  fprintf(stream, "  Short history: %d bits\n",
          pred->adaptive.gshare_short->config.two.shift_width);
  fprintf(stream, "  Long history : %d bits\n",
          pred->adaptive.gshare_long->config.two.shift_width);
  fprintf(stream, "  Chooser entries: %d\n",
          pred->adaptive.chooser->config.bimod.size);
  break;
```

This block prints the configuration details of the **Adaptive Gshare predictor** when the simulator starts.

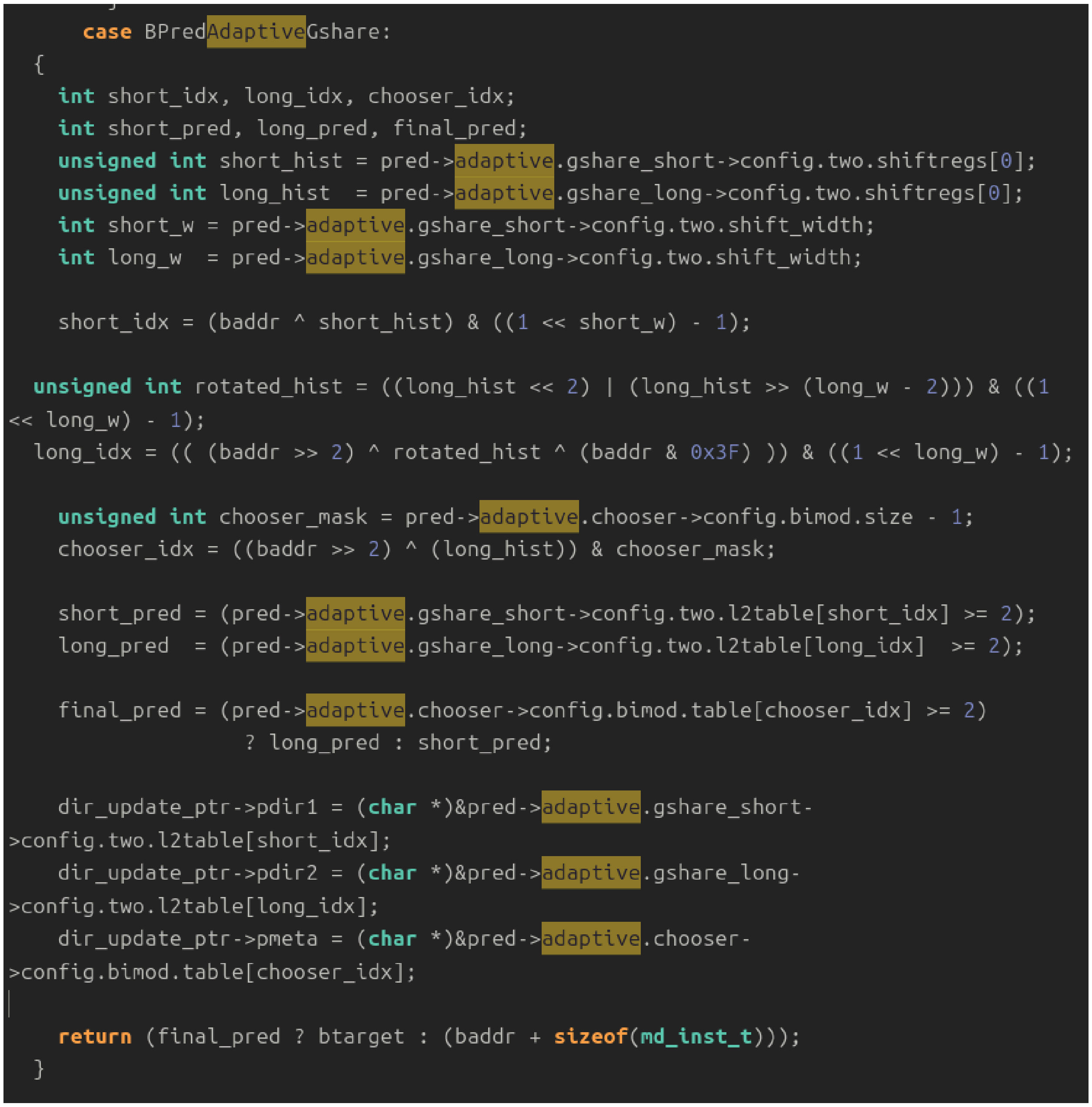
C) In bpred_lookup():

This block performs the **prediction phase** of the Adaptive Gshare predictor.

It computes indices for the **short**, **long**, and **chooser** tables using the program counter (PC) and corresponding history registers.

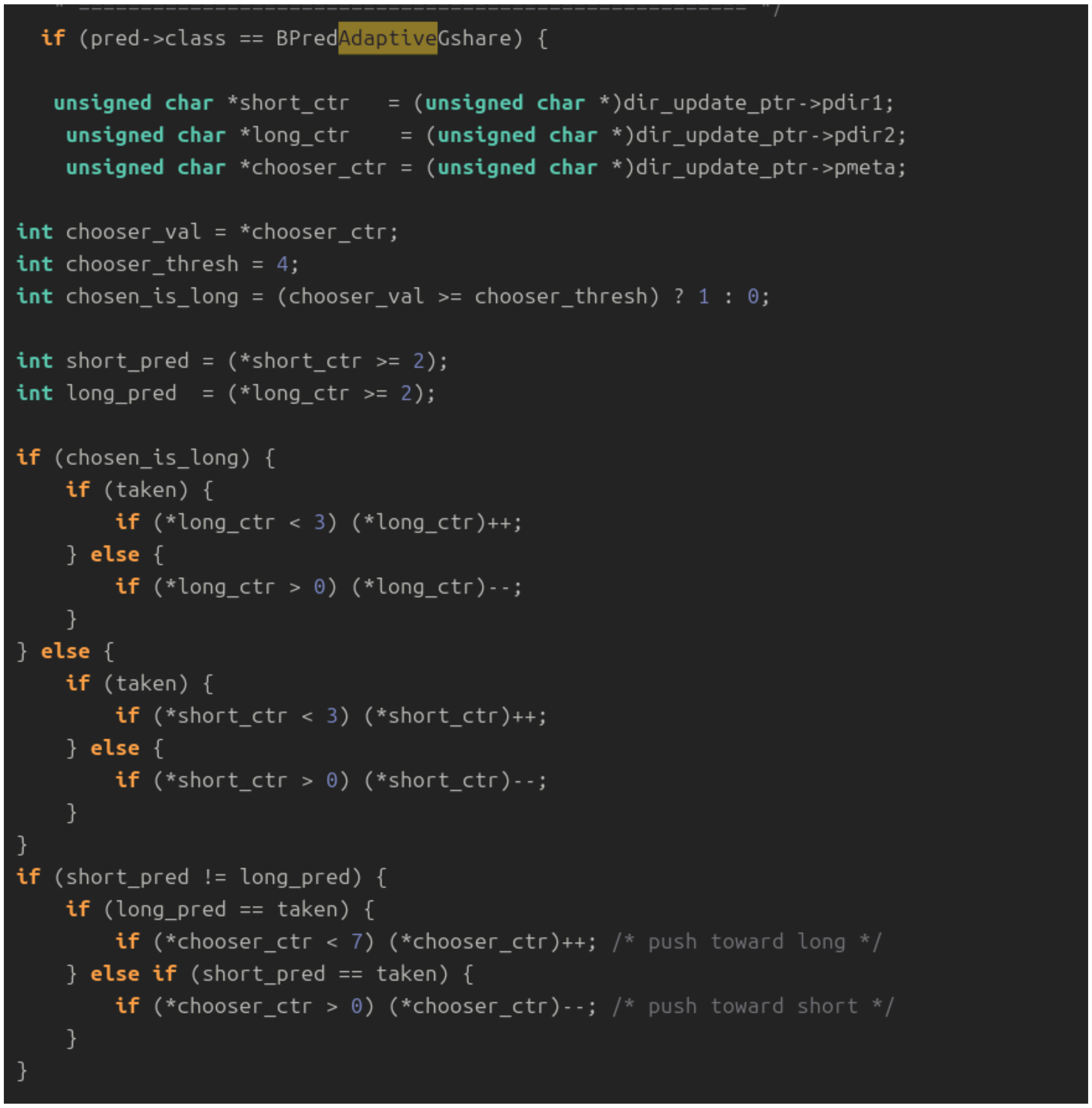
Each Gshare predictor generates its own prediction based on 2-bit counters, and the chooser decides which one to trust.

Finally, the selected prediction (`final_pred`) determines the next instruction address — either the predicted branch target or the sequential address.



```
      case BPredAdaptiveGshare:
  {
    int short_idx, long_idx, chooser_idx;
    int short_pred, long_pred, final_pred;
    unsigned int short_hist = pred->adaptive.gshare_short->config.two.shiftregs[0];
    unsigned int long_hist  = pred->adaptive.gshare_long->config.two.shiftregs[0];
    int short_w = pred->adaptive.gshare_short->config.two.shift_width;
    int long_w  = pred->adaptive.gshare_long->config.two.shift_width;

    short_idx = (baddr ^ short_hist) & ((1 << short_w) - 1);

  unsigned int rotated_hist = ((long_hist << 2) | (long_hist >> (long_w - 2))) & ((1
<< long_w) - 1);
  long_idx = (( (baddr >> 2) ^ rotated_hist ^ (baddr & 0x3F) )) & ((1 << long_w) - 1);

    unsigned int chooser_mask = pred->adaptive.chooser->config.bimod.size - 1;
    chooser_idx = ((baddr >> 2) ^ (long_hist)) & chooser_mask;

    short_pred = (pred->adaptive.gshare_short->config.two.l2table[short_idx] >= 2);
    long_pred  = (pred->adaptive.gshare_long->config.two.l2table[long_idx]  >= 2);

    final_pred = (pred->adaptive.chooser->config.bimod.table[chooser_idx] >= 2)
                   ? long_pred : short_pred;

    dir_update_ptr->pdir1 = (char *)&pred->adaptive.gshare_short-
>config.two.l2table[short_idx];
    dir_update_ptr->pdir2 = (char *)&pred->adaptive.gshare_long-
>config.two.l2table[long_idx];
    dir_update_ptr->pmeta = (char *)&pred->adaptive.chooser-
>config.bimod.table[chooser_idx];

    return (final_pred ? btarget : (baddr + sizeof(md_inst_t)));
  }
```

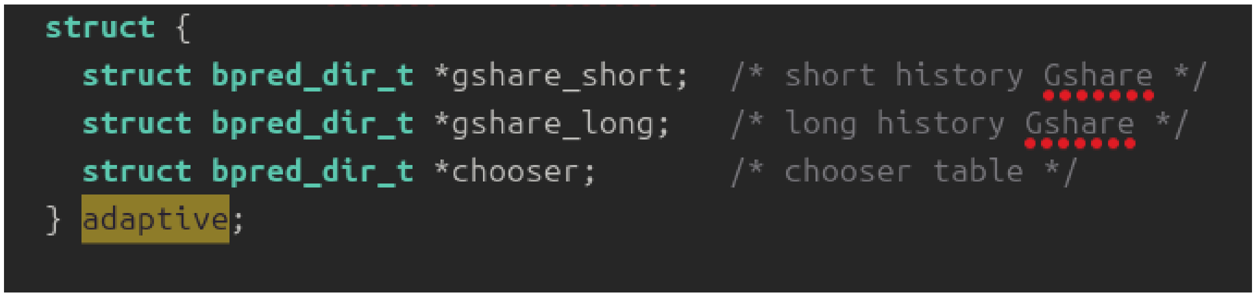
D) In brped_update():



```
  if (pred->class == BPredAdaptiveGshare) {

    unsigned char *short_ctr   = (unsigned char *)dir_update_ptr->pdir1;
     unsigned char *long_ctr    = (unsigned char *)dir_update_ptr->pdir2;
     unsigned char *chooser_ctr = (unsigned char *)dir_update_ptr->pmeta;

int chooser_val = *chooser_ctr;
int chooser_thresh = 4;
int chosen_is_long = (chooser_val >= chooser_thresh) ? 1 : 0;

int short_pred = (*short_ctr >= 2);
int long_pred  = (*long_ctr >= 2);

if (chosen_is_long) {
    if (taken) {
        if (*long_ctr < 3) (*long_ctr)++;
    } else {
        if (*long_ctr > 0) (*long_ctr)--;
    }
} else {
    if (taken) {
        if (*short_ctr < 3) (*short_ctr)++;
    } else {
        if (*short_ctr > 0) (*short_ctr)--;
    }
}
if (short_pred != long_pred) {
    if (long_pred == taken) {
        if (*chooser_ctr < 7) (*chooser_ctr)++; /* push toward long */
    } else if (short_pred == taken) {
        if (*chooser_ctr > 0) (*chooser_ctr)--; /* push toward short */
    }
}
```

This block updates the **Adaptive Gshare predictor** after each branch outcome is known. It adjusts the 2-bit counters of the **chosen predictor** (short or long) based on whether the branch was taken or not, allowing it to learn over time.The **chooser counter** is updated only when the two predictors disagree — rewarding the predictor that was correct and penalizing the one that was wrong. This selective update mechanism helps the chooser specialize and improve overall prediction accuracy.

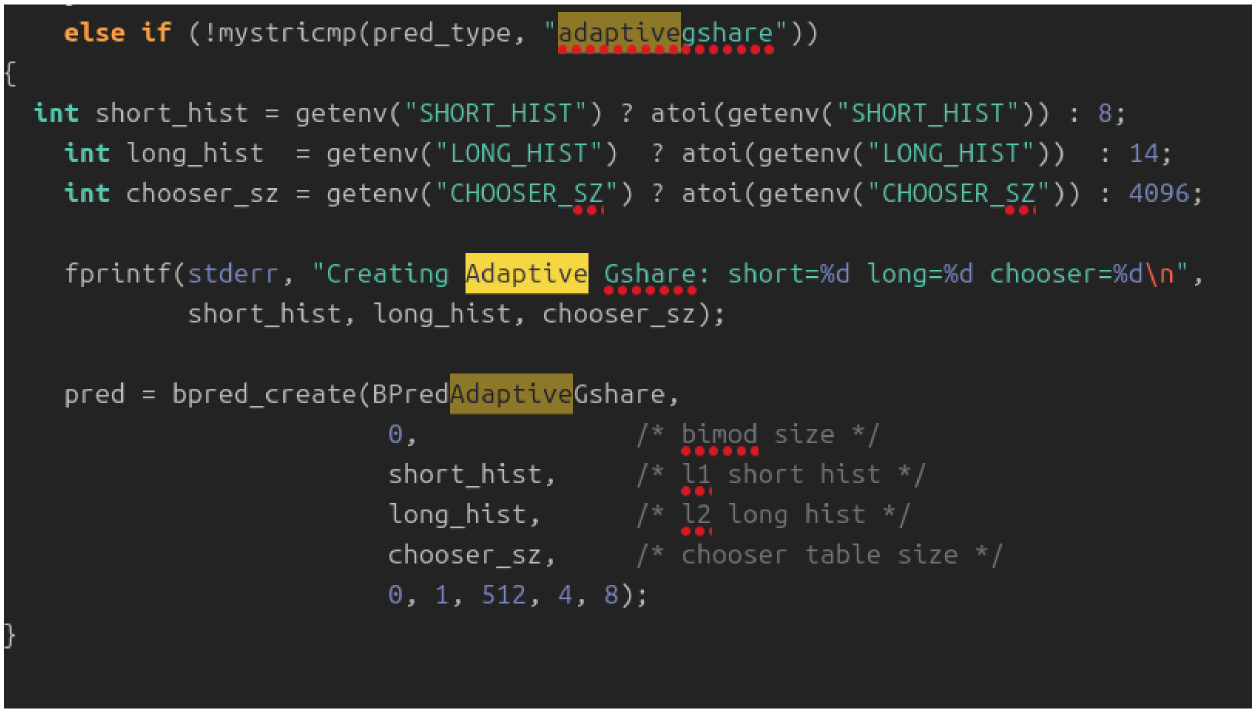
BPRED.H



```
struct {
  struct bpred_dir_t *gshare_short;   /* short history Gshare */
  struct bpred_dir_t *gshare_long;    /* long history Gshare */
  struct bpred_dir_t *chooser;        /* chooser table */
} adaptive;
```

This structure defines the components of the **Adaptive Gshare predictor** inside the branch predictor class.It contains pointers to three sub-predictors — the **short-history Gshare**, **long-history Gshare**, and the **chooser table**.

SIM-BPRED.C



```
  else if (!mystricmp(pred_type, "adaptivegshare"))
{
 int short_hist = getenv("SHORT_HIST") ? atoi(getenv("SHORT_HIST")) : 8;
  int long_hist  = getenv("LONG_HIST")  ? atoi(getenv("LONG_HIST"))  : 14;
  int chooser_sz = getenv("CHOOSER_SZ") ? atoi(getenv("CHOOSER_SZ")) : 4096;

  fprintf(stderr, "Creating Adaptive Gshare: short=%d long=%d chooser=%d\n",
          short_hist, long_hist, chooser_sz);

  pred = bpred_create(BPredAdaptiveGshare,
                      0,              /* bimod size */
                      short_hist,     /* l1 short hist */
                      long_hist,      /* l2 long hist */
                      chooser_sz,     /* chooser table size */
                      0, 1, 512, 4, 8);
}
```

This block handles the command-line option for selecting the **Adaptive Gshare predictor** in the simulator.It reads the short history, long history, and chooser size parameters (either from environment variables or default values), prints the chosen configuration, and

Parameter Configuration:

The Adaptive Gshare predictor was configured with a **short history length of 6 bits**, a **long history length of 20 bits**, and a **chooser table of 16K entries**.

The short history allows the predictor to quickly adapt to branches with frequent direction changes, while the long history captures patterns that span across larger instruction windows.

A chooser size of 16K was selected as it provides a good balance between prediction accuracy and storage overhead — large enough to minimize aliasing without significantly increasing hardware cost.This configuration was found experimentally to give the best overall accuracy and lowest misprediction count among all tested combinations.

Experimental Setup

A) **Evaluation Metrics:**

The performance of the branch predictors was evaluated using the following metrics:

Address Prediction Rate: Ratio of correctly predicted branch target addresses to total branches.

Direction Prediction Rate: Ratio of correctly predicted branch directions (taken/not taken) to total branches.

Total Misses: Number of incorrect predictions made during program execution.

JR Rate (Jump Register Rate): Accuracy of predictions for jump and return instructions, indicating how well the predictor handles indirect branches.

B) **Comparison Baselines: (for test-fmath)**

Predictor	**Short**	**Long**	**Chooser**	**Addr Rate**	**Dir Rate**	**Misses**
Gshare	4	20	8192	85.48	88.45	1212

C) Results and Discussion

For test-fmath

Predictor	**Short**	**Long**	**Chooser**	**Addr Rate**	**Dir Rate**	**Misses**
Adaptive	6	18	16384	85.46	85.46	1152
Adaptive	6	20	16384	85.89	85.89	1118
Adaptive	8	24	32768	85.89	85.89	1118
Adaptive	10	24	32768	85.22	85.22	1171
Adaptive	12	30	32768	85.36	85.36	1160

For Anagram(It didn’t perform good)

Gshare: Addr Rate: 88.7, Misses: 3833

Predictor	**Short**	**Long**	**Chooser**	**Addr Rate**	**Misses**
Adaptive	8	14	4096	87.65	4785
Adaptive	10	16	8192	86.89	5,080
Adaptive	10	14	8192	85.07	4824
Adaptive	10	16	8192	86.3	4427

PART2: ADAPTIVE HYBRID PREDICTOR(BY- VARRI NAVYA)

Simulator Setup

The project was implemented using the **SimpleScalar 3.0** simulator, a widely used toolset for studying microprocessor architecture and branch prediction mechanisms.

Benchmarks -test-math

Design of Adaptive Gshare Predictor:

The Hybrid Branch Predictor combines two prediction mechanisms — bimodal and two-level adaptive (2lev) — and uses a meta predictor (also called a *combining* or *chooser table*) to decide which one to trust for each branch.

Bimodal Table Size: Number of 2 bit counters used in the   bimodal predictor

L1 Size (History Registers): Number of history registers used in the 2_level predictor

L2 Size (Pattern Table): Number of 2 bit counters in the 2_level pattern table

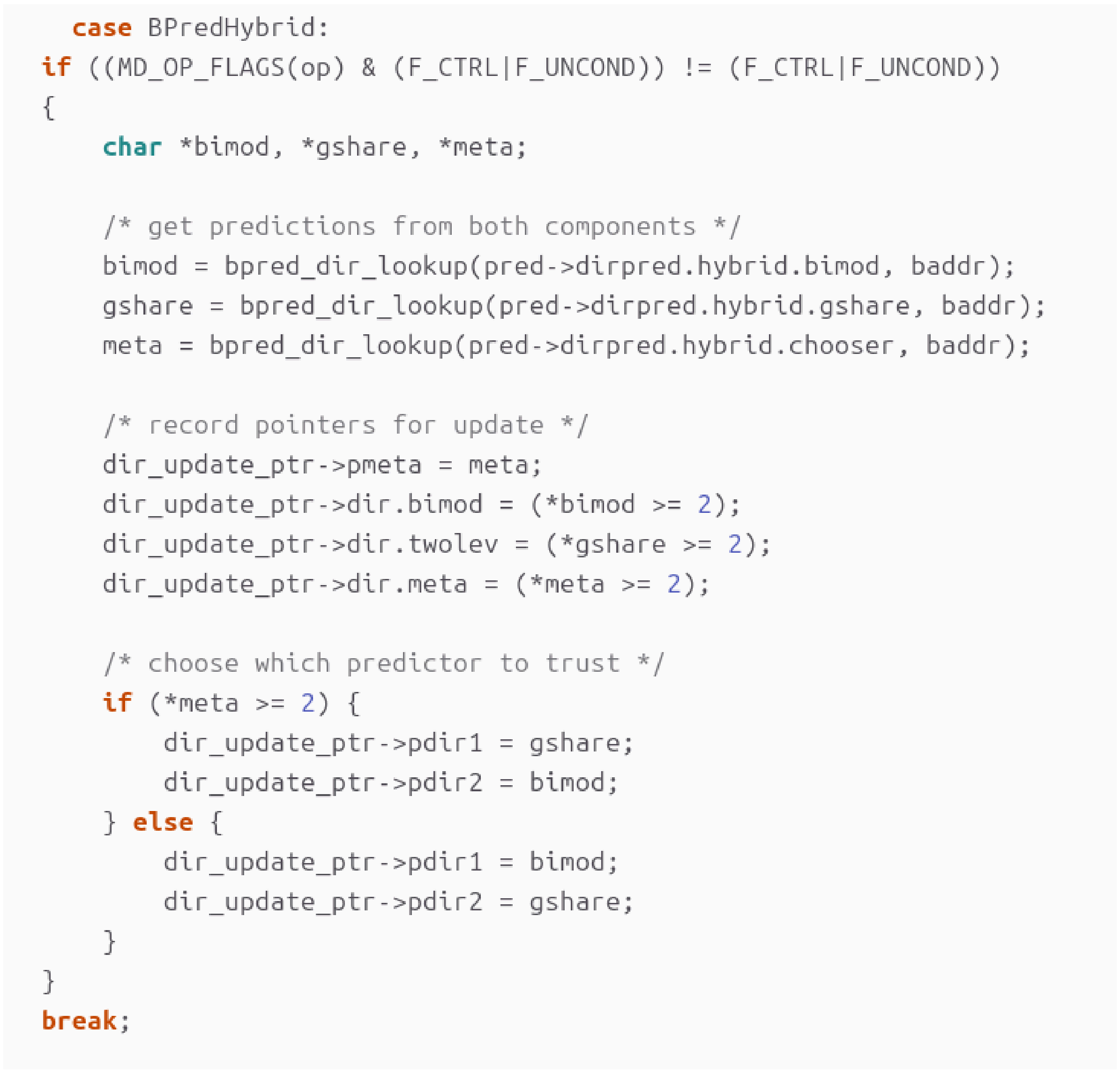
History Length: Number of bits in each history register

XOR Flag: whether to XOR the history bits with PC bits for indexing

Meta (Combining Table) Size: Number of 2-bit counters in the meta predictor used to choose between bimod and 2lev

Integration with SimpleScalar:

Changes in bpred.c: `BPredHybrid:` block in bpred.c



```
  case BPredHybrid:
if ((MD_OP_FLAGS(op) & (F_CTRL|F_UNCOND)) != (F_CTRL|F_UNCOND))
{
    char *bimod, *gshare, *meta;

    /* get predictions from both components */
    bimod = bpred_dir_lookup(pred->dirpred.hybrid.bimod, baddr);
    gshare = bpred_dir_lookup(pred->dirpred.hybrid.gshare, baddr);
    meta = bpred_dir_lookup(pred->dirpred.hybrid.chooser, baddr);

    /* record pointers for update */
    dir_update_ptr->pmeta = meta;
    dir_update_ptr->dir.bimod = (*bimod >= 2);
    dir_update_ptr->dir.twolev = (*gshare >= 2);
    dir_update_ptr->dir.meta = (*meta >= 2);

    /* choose which predictor to trust */
    if (*meta >= 2) {
        dir_update_ptr->pdir1 = gshare;
        dir_update_ptr->pdir2 = bimod;
    } else {
        dir_update_ptr->pdir1 = bimod;
        dir_update_ptr->pdir2 = gshare;
    }
}
break;
```

This **BPredHybrid block** is added to implement the **hybrid branch predictor logic**. It lets the simulator fetch predictions from both the **bimodal** and **gshare/2-level** predictors, then use a **meta (chooser) table** to decide which one to trust.This addition enables **dynamic selection between predictors**, improving overall branch prediction accuracy.

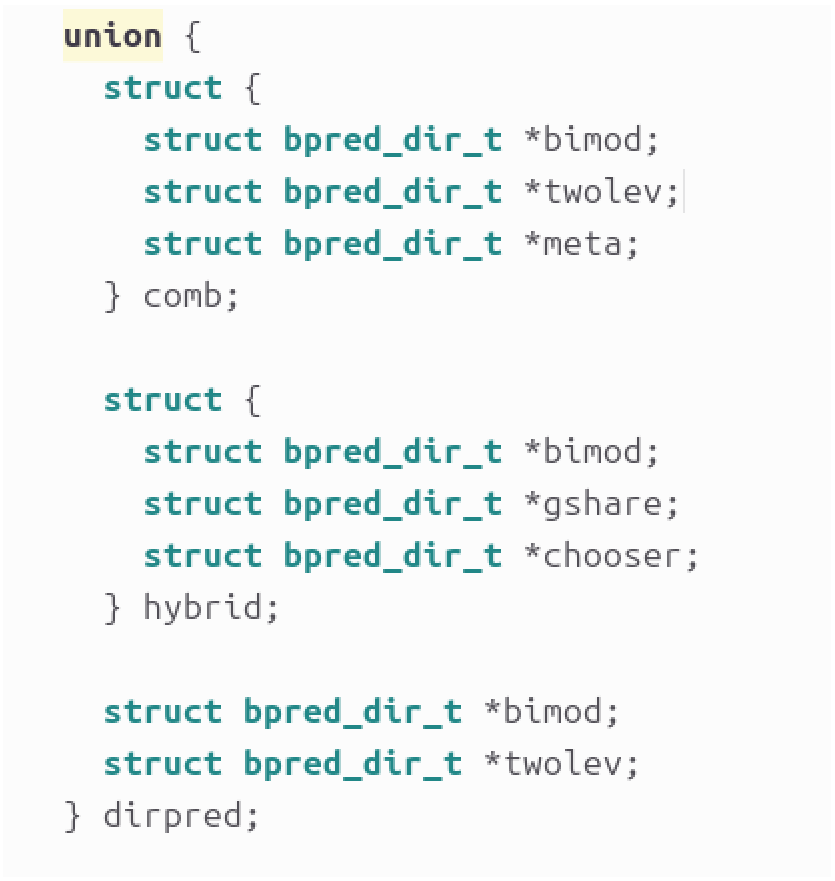
```
   switch (pred->class)
{
    case BPredComb:
      name = "bpred_comb";
      break;
    case BPred2Level:
      name = "bpred_2lev";
      break;
    case BPred2bit:
      name = "bpred_bimod";
      break;
    case BPredTaken:
      name = "bpred_taken";
      break;
    case BPredNotTaken:
      name = "bpred_nottaken";
      break;
    case BPredHybrid:
      name = "bpred_hybrid";
      break;
    default:
      panic("bogus branch predictor class");
}
```

I added the `BPredHybrid` **case** to ensure the simulator recognizes and correctly labels the new **hybrid predictor type**.

It assigns the name `"bpred_hybrid"` for reporting and statistics output.

Without this addition, the simulator would not identify or print results for the hybrid predictor.

Changes in bpred.h:



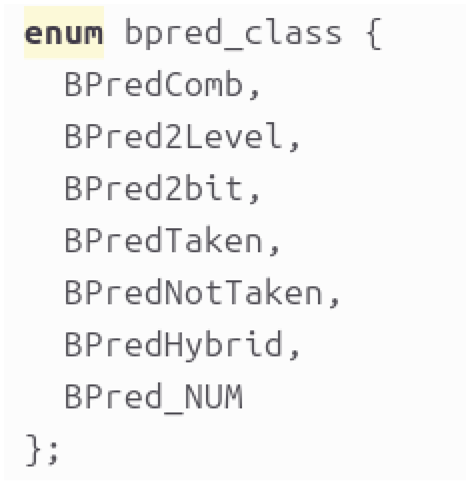
```
union {
  struct {
    struct bpred_dir_t *bimod;
    struct bpred_dir_t *twolev;
    struct bpred_dir_t *meta;
  } comb;

  struct {
    struct bpred_dir_t *bimod;
    struct bpred_dir_t *gshare;
    struct bpred_dir_t *chooser;
  } hybrid;

  struct bpred_dir_t *bimod;
  struct bpred_dir_t *twolev;
} dirpred;
```

I added the **`hybrid`** **structure** in `bpred.h` to define the three components of the hybrid predictor — **bimod**, **gshare**, and **chooser**.

This allows the simulator to store and access each sub-predictor separately within one unified predictor object.

Without this, the hybrid predictor couldn’t link its internal prediction tables or function correctly.

Adding BPredHybrid in both in bpred.c and bpred.h:

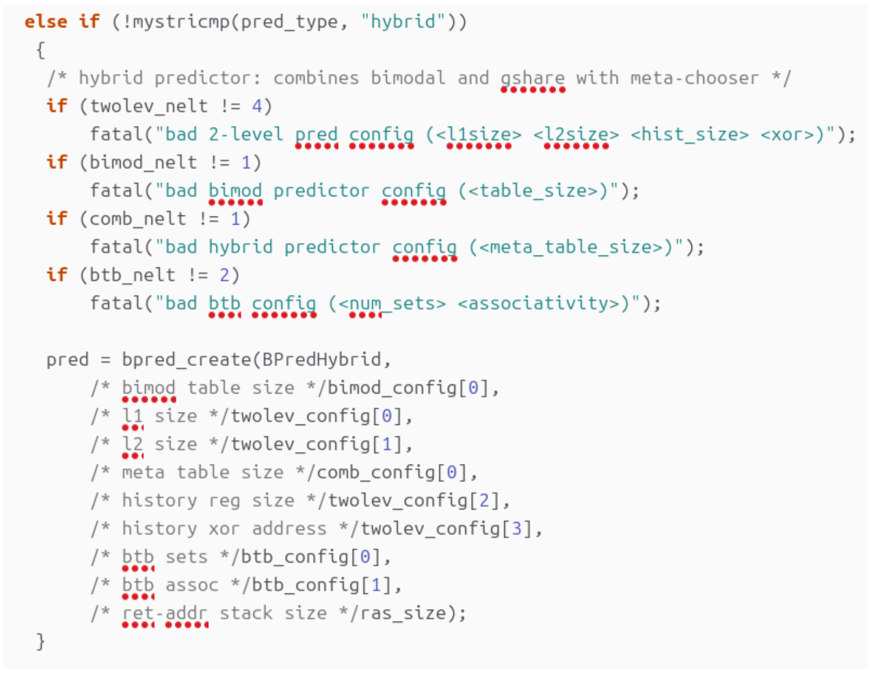


```
enum bpred_class {
  BPredComb,
  BPred2Level,
  BPred2bit,
  BPredTaken,
  BPredNotTaken,
  BPredHybrid,
  BPred_NUM
};
```

I added `BPredHybrid` to the `enum bpred_class` so the simulator can recognize the **hybrid predictor** as a distinct prediction type.This lets `bpred.c` handle hybrid-specific logic separately from other predictors.

Without it, the simulator wouldn't identify or execute the hybrid predictor configuration.

In simbpred.c:



```
else if (!mystricmp(pred_type, "hybrid"))
 {
  /* hybrid predictor: combines bimodal and gshare with meta-chooser */
  if (twolev_nelt != 4)
      fatal("bad 2-level pred config (<l1size> <l2size> <hist_size> <xor>)");
  if (bimod_nelt != 1)
      fatal("bad bimod predictor config (<table_size>)");
  if (comb_nelt != 1)
      fatal("bad hybrid predictor config (<meta_table_size>)");
  if (btb_nelt != 2)
      fatal("bad btb config (<num_sets> <associativity>)");

  pred = bpred_create(BPredHybrid,
      /* bimod table size */bimod_config[0],
      /* l1 size */twolev_config[0],
      /* l2 size */twolev_config[1],
      /* meta table size */comb_config[0],
      /* history reg size */twolev_config[2],
      /* history xor address */twolev_config[3],
      /* btb sets */btb_config[0],
      /* btb assoc */btb_config[1],
      /* ret-addr stack size */ras_size);
 }
```

I added this hybrid block in `sim-bpred.c` to enable command-line support for the new Hybrid Predictor type.

It validates user inputs and passes all configuration parameters to `bpred_create()` for initializing bimod, gshare, and meta components.

Without this, the simulator couldn't recognize or run the `-bpred hybrid` option.

Parameter Configuration:

Bimodal Table: 2048 entries

2-Level Predictor (2lev): L1 = 1 (global history), L2 = 8192 entries, History length = 12 bits, XOR = 1

Meta (Combining Table): 2048 entries

BTB (Branch Target Buffer): 512 sets × 4-way associativity

Return Address Stack (RAS): 8 entries

Experimental Setup:

A) Simulation Parameters:
Benchmarks: `test-math`, `test-float`, `test-fmath`, `test-mem`, `test-branch` (binaries in `./tests-pisa/bin.little/`).

Baseline (2lev-only):

./sim-bpred -bpred 2lev \-bpred:2lev 1 2048 10 0 \ ./tests-pisa/bin.little/test-math \
> results/baseline_2lev_testmath.log

hybrid :
/sim-bpred -bpred hybrid \-bpred:bimod 2048 \-bpred:2lev 1 8192 12 1
\-bpred:comb 2048 \./tests-pisa/bin.little/test-fmath \>
results/hybrid_testfmath_8192_12_2048.log

D) **Evaluation Metrics:**

`sim_num_insn` — total instructions executed (sanity check)

`sim_num_branches` — total branches executed.

`bpred_hybrid.lookups` / `bpred_*.lookups` — branch lookups

`bpred_hybrid.updates` — predictor updates

`bpred_hybrid.addr_hits` — address (target) hits.

`bpred_hybrid.dir_hits` — direction hits (correct taken/not-taken).

`bpred_hybrid.misses` — total mispredictions

E) Comparison Baselines:

- Bimodal-only (`-bpred bimod -bpred:bimod 2048`) to compare simple predictor behavior.

- **Hybrid** (your tuned hybrid) to show best achieved accuracy.

	Bimod Size	L1	L2	Hist	XOR	Meta Size	Addr Rate	Dir Rate	Misses	RAS Rate
standard	2048	–	–	–	–	–	0.886	0.9007	3196	99.97

F) Results and Discussion

	Bimod Size	L1	L2	Hist	XOR	Meta Size	Addr Rate	Dir Rate	Misses	RAS Rate
Hybrid test_math	2048	1	2048	10	1	1024	88.61	90.05	3202	99.97
test_math	2048	1	2048	12	1	1024	88.6	90.07	3196	99.97
test_math	2048	1	4096	12	1	1024	88.93	90.39	3091	99.97
test_math	2048	1	8192	12	1	2048	88.93	90.4	3090	99.97
test_math	2048	256	8192	12	1	2048	88.93	90.4	3090	99.97
test_fmath	2048	1	8192	12	1	2048	86.01	89.53	815	99.84

Conclusion

In this project, two hybrid branch prediction schemes were implemented and evaluated using the SimpleScalar simulator. One model combined **two Gshare predictors** of different history lengths (Adaptive Gshare), while the other integrated a **Bimodal and Two-Level predictor** (Hybrid Bimod+2lev).

Both predictors were tested on different benchmarks — *test-fmath* and *test-math* — and showed improved performance compared to the baseline predictors provided by SimpleScalar.

The Adaptive Gshare achieved a lower misprediction count and higher address prediction rate, while the Hybrid Bimod+2lev delivered strong direction accuracy.

These results demonstrate that hybrid designs effectively balance short-term and long-term branch behavior, leading to overall better prediction accuracy with minimal hardware overhead.