MODIFIED CODE PART EXPLANATION

A. Initialize counter in two table

The SimpleScalar initialize counters in two-level adaptive predictor and bimodal predictor to weakly this-or-that module. Which means even entries (0,2,4,6…) store 1 (01 weakly not-taken) and odd entries (1,3,5,7…) store 2 (10 weakly-taken) at first.

```
/* initialize counters to weakly this-or-that */
flipflop = 1;
for (cnt = 0; cnt < l2size; cnt++)
{
pred_dir->config.two.l2table[cnt] = flipflop;
flipflop = 3 - flipflop;
}
```

Figure 1. Original counter setting.

In the two-mode predictor, we still use the similar mode, weakly this-or-that, but as we use a 4-bit counter instead of 2, we need to change the weakly not-taken to 0111 and weakly taken to 1000 at first. So, the counters store in entries should be (7, 8, 7, 8…).

The GShare global two-level predictor decide the first 2 bit of 4-bit counter, so it will keep the same setting as SimpleScalar as before, (1,2,1, 2...).

```
/* initialize counters to weakly this-or-that */
flipflop = 1;
for (cnt = 0; cnt < l2size; cnt++)
{
pred_dir->config.two.l2table[cnt] = flipflop;
flipflop = 3 - flipflop;
}
```

Figure 2. Gshare branch predictor counter initialization.

For the new-designed mode, it generates the last 2 bits of the 4-bit counter, and the initialization result should be 7, 8, 7, 8…, so the odd counter should be 3 (11) and the even counter should be 0 (00). So, the new initialization code of it shown in Figure 3. The first entry stores a counter value -1, it is a single that no correlated branches are include in such hashed address, so -1 tell the two-mode predictor that new-designed predictor will not be used.

```
flipflop = 3;
for (cnt = 0; cnt < l1size; cnt++)
  {
pred_dir->config.bimod.table[cnt] = flipflop;
flipflop = 3 - flipflop;
  }
pred_dir->config.bimod.table[0] = -1;
```

Figure 3. New-designed predictor counter initialization.

B. Find entry in two table:

The Gshare algorithm about how to index a entry in PHT already coded in SimpleScalar.

The two-mode predictor also need a new hash algorithm to calculate the hash result to find the value of correlated branch predictor. But this algorithm will be changed for each program because we don't know which address represent such correlated branches and others. So, in the code we design, we assume that all branch address ended with eight 0s will equal to the hashed result. For example, 00000000 represents non-correlated branches, and 10010010 represents such branch is correlated by B_1, $!B_2$ and B_4.

```
#define BIMOD_HASH(PRED, ADDR)                          \
  ((((ADDR) >> 19) ^ ((ADDR) >> MD_BR_SHIFT)) & ((PRED)->config.bimod.size-1))
    /* was: ((baddr >> 16) ^ baddr) & (pred->dirpred.bimod.size-1) */
```

Figure 4. Hash algorithm.

C. Get counter in entry, combine and prediction:

Figure 5 shows how to get two counters in two table, and we delete the meta-predictor, and we will use both predictor in each time (except all 0 situation). So, predictor one will always be 2-level predictor and the second predictor will always be new-designed predictor.

```
bimod = bpred_dir_lookup (pred->dirpred.bimod, baddr);
twolev = bpred_dir_lookup (pred->dirpred.twolev, baddr);

dir_update_ptr->dir.bimod = (*bimod >= 2);
dir_update_ptr->dir.twolev  = (*twolev >= 2);

    dir_update_ptr->pdir1 = twolev;
    dir_update_ptr->pdir2 = bimod;
```

Figure 5. Find counter in two table.

Figure 6 shows how to combine two counters together and predict the result. If the counter store in new-designed predictor is -1, that means no correlated predictor contained in such branch, so we only use 2-level predictor to prediction. But when it is not -1, we will calculate the new 4-bit counter and predict taken if it is equal or greater than 8. (1000)

```
   if(*(dir_update_ptr->pdir2)==-1)
{
  /* BTB miss -- just return a predicted direction */
  return ((*(dir_update_ptr->pdir1) >= 2)
     ? /* taken */ 1
     : /* not taken */ 0);
}
else
{
   int a=*dir_update_ptr->pdir1<<2|*dir_update_ptr->pdir2;
   return ((a >= 8)
     ? /* taken */ 1
     : /* not taken */ 0);
}
```

Figure 6. Combine counter and make prediction

D. Update 4-bit counter:

Figure 7 shows how to update the 4-bit counter and store the result into two tables separately. If the prediction result is right, the 4-bit counter will add 1, can't extend 15 (1111); if the result is wrong, the 4-bit counter will minus 1, no less than 0 (0000). After update the 4-bit counter, we separate new counter into two 2-bit counter and update the counter in the Pattern History Table and Pattern Table to finish this prediction action.

```
  if (dir_update_ptr->pdir1 && *dir_update_ptr->pdir2!=-1)
{int a = *dir_update_ptr->pdir1 <<2 | *dir_update_ptr->pdir2;
  if (taken)
{
  if (a <15)
   a++;
}
  else
{ /* not taken */
  if (a> 0)
    a--;
}
*dir_update_ptr->pdir1=a>>2;
*dir_update_ptr->pdir2=a^ ((a>>2)<<2);
}
```

Figure 7. Combine counter and make prediction

E. Initialization:

To simple the testing and running part of two-mode predictor, I initialize the comb module in SimpleScalar to create the two-mode predictor included a GShare two-level adaptive branch predictor, with a one-l1-size, 10-history-register-size, and 1024-Pattern-History-Table size, combined with a 2048-size Pattern Table new designed predictor.

The comand in linus should be ./sim-bpred -bpred comb test-program-name.

```
/* new designed predictor config (<table_size>) */
static int bimod_nelt = 1;
static int bimod_config[1] =
  { /* new tbl size */2048 };

/* 2-level predictor config (<l1size> <l2size> <hist_size> <xor>) */
static int twolev_nelt = 4;
static int twolev_config[4] =
  { /* l1size */1, /* l2size */1024, /* hist */10, /* xor */TRUE};
```

Figure 8. Two-mode predictor initialization.