Technical Report UEC-IS-2003-2, Version 2003-05-06
Graduate School of Information Systems, University of Electro-Communications

高性能プロセッサのための代表的な分岐予測器の実装と評価

吉瀬 謙二　片桐 孝洋　本多 弘樹　弓場 敏嗣
電気通信大学 大学院情報システム学研究科

命令発行幅の増大と命令パイプライン長の増大により，プロセッサ性能に与える分岐予測器の重要性が増している．予測精度を向上させるために，過去数十年の間に様々な分岐予測器が提案されている．本プロジェクトの目的は，高級言語を用いた実装を示すことで代表的な分岐予測器の挙動を明確にすること．及び，ベンチマークプログラムを用いて代表的な分岐予測器の性能を評価することにある．高性能プロセッサのための代表的な分岐予測器として，Bimodal 分岐予測器，Gshare 分岐予測器，Bimode 分岐予測器を取り上げ，C++を用いてこれらの分岐予測器を実装する．また，ベンチマークプログラム SPEC CINT95 を用いて実装した分岐予測器を評価する．

キーワード 高性能プロセッサ, bimodal 分岐予測器, gshare 分岐予測器, bimode 分岐予測器

Implementation of Typical Branch Predictors for High Performance Microprocessors

Kenji Kise, Takahiro Katagiri, Hiroki Honda, and Toshitsugu Yuba

Graduate School of Information Systems,
University of Electro-Communications

Accurate branch prediction has come to play an important role for modern microprocessors. In order to improve its prediction accuracy, many branch predictors have been investigated in the past few decades. The aim of this project is to implement typical branch predictors to clarify the behavior of branch predictors. We implement Bimodal, Gshare and Bimode branch predictor in C++ from scratch. We evaluate typical branch predictors with the benchmark programs of SPEC CINT95.

Key-words high performance processor, bimodal predictor, gshare predictor, bimode predictor

1 はじめに

近年のマイクロプロセッサは，命令発行の幅を広くして命令レベル並列性を抽出し，命令パイプラインの段数を深くすることで動作周波数を向上させている．これらの傾向は，プロセッサ内で処理される命令数を増加させるため，分岐予測ミスが発生した際にフラッシュすべき命令数，すなわち分岐予測ミスのペナルティも増加する．これらミスペナルティの影響を軽減するために，精度の高い分岐予測器が高性能プロセッサを実現する上で不可欠となっている．

本プロジェクトの目的は，高級言語を用いた実装を示すことで代表的な分岐予測器の挙動を明確にすること．及び，ベンチマークプログラムを用いて代表的な分岐予測器の性能を評価することにある．高性能プロセッサのための代表的な分岐予測器として，Bimodal 分岐予測器，Gshare 分岐予測器，Bimode 分岐予測器を取り上げ，C++を用いてこれらの分岐予測器を実装する．また，ベンチマークプログラム SPEC CINT95 を用いて実装した分岐予測器を評価する．

本稿の構成を示す．2 章で幾つかの分岐予測器の実装を示す．3 章で実装した分岐予測器を評価する．4 章で本稿をまとめる．

2 分岐予測器の実装

本章では，代表的な分岐予測器として Bimodal, Gshare, Bimode の 3 つの分岐予測器の実装を示す．本節で示すソースコードは，C++を用いてゼロから記述したものである．

本稿では，分岐予測器のことを省略して予測器と呼ぶ

ことがある．また，分岐成立のことを taken，分岐不成立のことを untaken と呼ぶことがある．

分岐予測の詳細な調査は本稿の目的ではない．これらは，幾つかの文献 [2, 8, 9] に詳しい．

ソースコード中で利用される定数の定義を示す．

```
#define BUF_SIZE 4096
#define B_TKN 1 /* Branch Taken   */
#define B_UNT 0 /* Branch Untaken */
```

全ての分岐予測器は次に示すクラス BP (Branch Predictor) から派生する．このクラスは，予測成功回数とミス回数の計測や，評価結果の表示といった共通の機能を提供する．クラス BP のコードを示す．

```
class BP{ /*** Branch Prediction ***/
  int size; /* Hardware budget in bit */
  char name[BUF_SIZE];
  uint64_t num_of_hit, num_of_mis;
public:
  BP();  /* Constructor */
  ~BP(); /* Destructor  */
  void set_config(char*, int);
  void bookkeep(int, int);
};

BP::BP(){num_of_hit = num_of_mis = 0;}

BP::~BP(){
  printf("===\n=== BP_Name: %s, ", name);
  printf("%9.2f KB\n", (float)size/8192.0);
  printf("=== Sum HIT MISS HITrate: ");
  printf("%011lld: %011lld %011lld: ",
         num_of_hit + num_of_mis,
         num_of_hit, num_of_mis);
  printf("%06.2f\n",
         (double)num_of_hit * 100.0 /
         (double)(num_of_hit + num_of_mis));
}

void BP::set_config(char *n, int s){
  size = s;
  strncpy(name, n, BUF_SIZE);
}

void BP::bookkeep(int pred, int result){
  if(pred==result) num_of_hit++;
  else             num_of_mis++;
}
```

2.1 2 ビット飽和型カウンタ

2 ビット飽和型カウンタは様々な分岐予測器の構成要素として利用される．

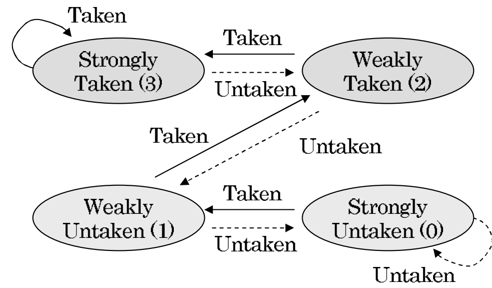

図 1: 2 ビット飽和型カウンタの状態遷移

2 ビット飽和型カウンタの状態遷移を図 1 に示す．2 ビットで表現できる 3,2,1,0 という 4 つの状態は strongly taken, weakly taken, weakly untaken, strongly untaken と呼ばれる．2 ビットのカウンタは，分岐結果が成立の時にインクリメントして，分岐結果が不成立の時にデクリメントする．カウンタの値が 1 より大きい時に分岐成立と予測し，1 以下の時に分岐不成立と予測する．

2 ビット飽和型カウンタを実現するクラス TwoSC のコードを示す．

```
class TwoSC{ /** Two-bit Saturating Counter **/
  char cnt;
public:
  void init(char);
  void increment();
  void decrement();
  char load();
  int predict();
};

void TwoSC::init(char init){ cnt = init; }
char TwoSC::load(){ return cnt;};
void TwoSC::increment(){ if(cnt<3) cnt++; }
void TwoSC::decrement(){ if(cnt>0) cnt--; }
int TwoSC::predict(){
  if(cnt>1) return B_TKN; else return B_UNT;
}
```

2.2 Bimodal 予測器

Smith[5] により提案された Bimodal 予測器は，予測精度がそれほど高くないために単体として利用されることは少ないが，他の分岐予測器やハイブリッド分岐予測器の構成要素として広く利用されている．

Bimodal 予測器の構成を図 2 に示す．分岐命令のアドレス (プログラムカウンタ) の下位 n ビットを用いて，パターン履歴表 (PHT) を参照する．PHT は図 1 に示す 2

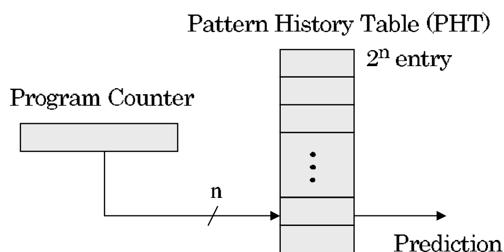

図 2: Bimodal 予測器のブロック図

ビット飽和型カウンタの配列であり，選択された 2 ビットカウンタの値により分岐方向を予測する．

2^n を PHT のエントリ数として，Bimodal 予測器のハードウェア量 (単位はビット) は次式で計算される．

$$2\ bit \times 2^n\ entry \quad (1)$$

Bimodal 予測器の実装を示す．PHT の初期値として weakly taken (2) を利用する．

```
class bimodal : public BP{
  int n;       /* n-bit                 */
  int entry;   /* PHT entry             */
  TwoSC *pht;  /* Pattern History Table */
public:
  bimodal(int); /* Constructor */
  void init();  /* Initialize  */
  void predict(uint64_t, int*);
  void regist(uint64_t, int, int);
};

bimodal::bimodal(int n_bit){
  n     = n_bit;
  entry = 1 << n_bit;
  pht = new TwoSC[entry];
  char buf[BUF_SIZE];
  sprintf(buf, "bimodal(%2d)", n_bit);
  set_config(buf, entry*2);
  init();
}

void bimodal::init(){
  for(int i=0; i<entry; i++)
    pht[i].init(2); /* Init by Weakly Taken*/
}

void bimodal::predict(uint64_t pc, int *pred){
  int index = (pc>>2) % entry;
  *pred = pht[index].predict();
}

void bimodal::regist(uint64_t pc, int r,
                     int pred){
  int index = (pc>>2) % entry;
```

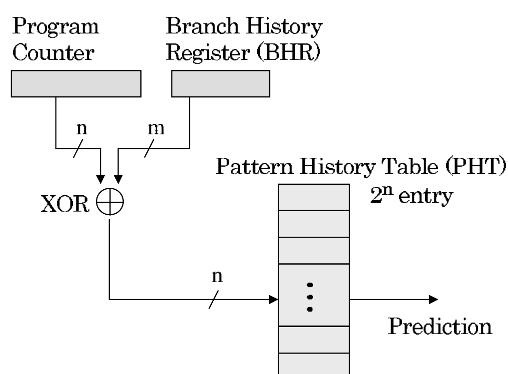

図 3: Gshare 予測器のブロック図

```
  if(r==B_TKN) pht[index].increment();
  if(r==B_UNT) pht[index].decrement();
  bookkeep(r, pred);
}
```

2.3 Gshare 予測器

McFarling ら [4] により提案された Gshare 予測器は，シンプルな構成で高い予測精度を達成するという特徴から，幾つかの商用プロセッサに採用されている．

Gshare 予測器の構成を図 3 に示す．これは，2 レベル適応型 [6] の分岐予測を拡張したもので，グローバル分岐履歴レジスタ (BHR) と分岐命令のアドレス (プログラムカウンタ) との排他的論理和によりパターン履歴表 (PHT) へのインデックスを作成する．PHT は 2 ビット飽和型カウンタの配列であり，選択された 2 ビットカウンタの値により分岐方向を予測する．

Gshare 予測器で利用されるインデックス関数の詳細を図 4 に示す．プログラムカウンタから切り出したビット長 n と分岐履歴レジスタのビット長 m とが異なる場合には，プログラムカウンタの上位ビットに揃えて排他的論理和を計算する点に注意する必要がある．

Gshare 予測器のハードウェア量は次式で計算される．ここで，2^n が PHT のエントリ数である．分岐履歴レジスタのハードウェア量は無視できる程小さいので，ここで示す式は Bimodal 予測器と同じものとなる．

$$2\ bit \times 2^n\ entry \quad (2)$$

Gshare 予測器の実装を示す．PHT の初期値として weakly taken (2) を利用する．

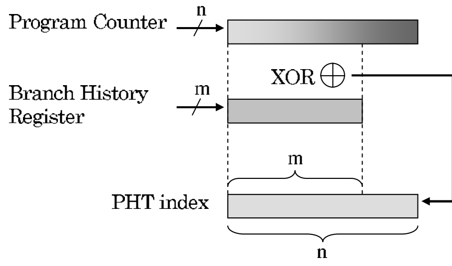

図 4: Gshare 予測器で用いられる排他的論理和によるインデックス関数

```
class gshare : public BP{
  int n,m;       /* n-bit & m-bit         */
  int entry;     /* PHT entry             */
  TwoSC *pht;    /* Pattern History Table */
 public:
  gshare(int,int); /* Constructor */
  void init();     /* Initialize  */
  void predict(uint64_t, uint64_t, int*);
  void regist(uint64_t, uint64_t, int, int);
};

gshare::gshare(int n_bit, int m_bit){
  n = n_bit;
  m = m_bit;
  entry = 1 << n_bit;
  if(m>n){printf("*** Error\n"); exit(1);}
  pht = new TwoSC[entry];
  char buf[BUF_SIZE];
  sprintf(buf, "gshare(%2d,%2d)", n_bit, m_bit);
  set_config(buf, entry*2);
  init();
}

void gshare::init(){
  for(int i=0; i<entry; i++)
    pht[i].init(2); /* Init by Weakly Taken*/
}

void gshare::predict(uint64_t pc, uint64_t bhr,
                     int *pred){
  int index = ((pc>>2) ^ (bhr<<(n-m))) % entry;
  *pred = pht[index].predict();
}

void gshare::regist(uint64_t pc, uint64_t bhr,
                    int r, int pred){
  int index = ((pc>>2) ^ (bhr<<(n-m))) % entry;
  if(r==B_TKN) pht[index].increment();
  if(r==B_UNT) pht[index].decrement();
  bookkeep(r, pred);
}
```

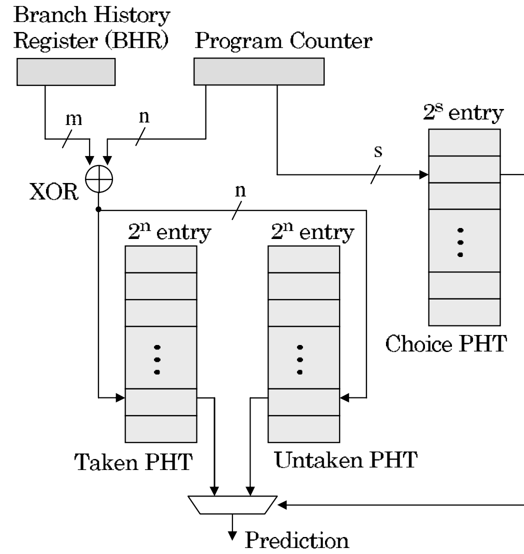

図 5: Bimode 予測器のブロック図

2.4 Bimode 予測器

Gshare 予測器の破壊的競合を緩和するために，Lee ら [3] により Bimode 予測器が提案された．同様の構成が Sgshare 予測器として野口ら [10] により提案されている．こちら方が投稿時期が早いことを考慮すると，本来は Sgshare 予測器と呼ぶべきである．しかしながら，Bimode 予測器として広く認知されているため，本稿では Bimode 予測器と記述する．

Bimode 予測器の構成を図 5 に示す．Bimode 予測器は Choice PHT, Taken PHT, Untaken PHT という 3 つのテーブルを利用する．ここで，Taken PHT と Untaken PHT のことを Direction PHT と呼ぶ．Choice PHT はプログラムカウンタによりインデックスを生成する 2 ビットカウンタのテーブルであり，Bimodal 予測器と同じ構成を持つ．分岐履歴レジスタ (BHR) と分岐命令のアドレスとの排他的論理和により Direction PHT のインデックスを生成する．Choice PHT が分岐成立と予測した場合には Taken PHT を，そうでない場合には Untaken PHT の値を用いて予測をおこなう．

Bimode 予測器のハードウェア量は次式で計算される．ここで，2^n は Direction PHT のエントリ数，2^s は Choice PHT のエントリ数である．

$$2\ bit \times 2^n\ entry \times 2 + 2\ bit \times 2^s\ entry \qquad (3)$$

Bimode 予測器の実装を示す．Taken PHT の初期値として weakly taken (2) を，Untaken PHT の初期値として

weakly untaken (1) を利用する．Choice PHT の初期値として weakly taken (2) を利用する．

```
class bimode : public BP{
  int n, m, s, n_entry, s_entry;
  TwoSC *choice;         /* choice predictor */
  TwoSC *pht_t;          /* Taken   PHT      */
  TwoSC *pht_u;          /* Untaken PHT      */
 public:
  bimode(int, int, int); /* Constructor */
  void init();            /* Initialize  */
  void predict(uint64_t, uint64_t, int*);
  void regist(uint64_t, uint64_t, int, int);
};

bimode::bimode(int n_bit, int m_bit, int s_bit){
  n = n_bit;
  m = m_bit;
  s = s_bit;
  n_entry = 1 << n_bit;
  s_entry = 1 << s_bit;
  choice= new TwoSC[s_entry];
  pht_t = new TwoSC[n_entry];
  pht_u = new TwoSC[n_entry];
  char buf[BUF_SIZE];
  sprintf(buf, "bi-mode(%2d,%2d,%2d)", n,m,s);
  set_config(buf, s_entry*2 + n_entry*4);
  init();
}

void bimode::init(){
  for(int i=0; i<n_entry; i++){
    pht_t[i].init(2); /* Weakly Taken   */
    pht_u[i].init(1); /* Weakly UnTaken */
  }
  for(int i=0; i<s_entry; i++)
    choice[i].init(2);/* Weakly Taken   */
}

void bimode::predict(uint64_t pc, uint64_t bhr,
                     int *pred){
  int index_c = (pc>>2) % s_entry;
  int index_p =((pc>>2) ^ (bhr<<(n-m))) % n_entry;

  if(choice[index_c].predict())
    *pred = pht_t[index_p].predict();
  else
    *pred = pht_u[index_p].predict();
}

void bimode::regist(uint64_t pc, uint64_t bhr,
                    int r, int pred){
  int index_c = (pc>>2) % s_entry;
  int index_p =((pc>>2) ^ (bhr<<(n-m))) % n_entry;

  if(choice[index_c].predict()){
    if(r==B_TKN) pht_t[index_p].increment();
    if(r==B_UNT) pht_t[index_p].decrement();
  }else{
    if(r==B_TKN) pht_u[index_p].increment();
    if(r==B_UNT) pht_u[index_p].decrement();
  }

  /** Partial update of the choice PHT **/
  TwoSC *cnt = &choice[index_c];
  if(r==cnt->predict() || r!=pred){
    if(r==B_TKN) cnt->increment();
    if(r==B_UNT) cnt->decrement();
  }
  bookkeep(r, pred);
}
```

2.5 評価のためのコード

2.5.1 SimAlpha への組み込み

実装した分岐予測器を評価するために SimAlpha[7] を利用する．SimAlpha は，ライブラリを静的にリンクしたアプリケーションのコードを実行する機能レベルのシミュレータである．

分岐命令を処理する際に，分岐予測の評価をおこなう関数 branch_predict の呼び出しを SimAlpha の中のメソッド Execute に追加する．変更を加えたコードを示す．167 行の関数呼び出しが追加されている．これを含めて，SimAlpha のコードへの変更点は数行と非常に少ない．

```
144 int instruction::Execute(data_t *Tpc){
145   /*** Update Rcv ***/
146   if(BR || Op==OP_JSR){
147     Rcv=Npc;
148   }
149   else if(!LD){
150     ALU(ir, &Rav, &Rbv, &Rcv);
151   }
152
153   /*** Update Adr ***/
154   Adr.init(0);
155   if(LD || ST){
156     ALU(ir, &Imm, &Rbv, &Adr);
157   }
158
159   /*** Update Tpc ***/
160   *Tpc = Npc;
161   if(Op==OP_JSR){
162     *Tpc = Rbv;
163     Tpc->st(Tpc->ld() & ~3ull);
164   }
165   if(BR){
166     BRU(ir, &Rav, &Rbv, &Npc, Tpc);
167     branch_predict(Npc.ld(), Tpc->ld());
168   }
169   return 0;
170 }
```

2.5.2 関数 branch_predict

複数の分岐予測器を評価したり，ある分岐予測器のパラメータを変更して評価をおこなう場合に，多くの分岐予測器の構成を同時に評価することができれば便利である．これを実現するための関数 branch_predict のコードを示す．

```
void branch_predict(uint64_t npc, uint64_t tpc){
  static uint64_t bhr=0; /* branch history reg */
  uint64_t pc = npc - 4; /* PC of the inst     */
  int result = npc!=tpc; /* branch taken ?     */

  static class EvaBimodal eva_bimodal;
  eva_bimodal.step(result, pc);

  static class EvaGshare eva_gshare;
  eva_gshare.step(result, pc, bhr);

  static class EvaBimode eva_bimode;
  eva_bimode.step(result, pc, bhr);

  bhr = (bhr << 1) | result;  /* Update BHR */
}
```

上に示したコードのクラスEvaBimodalによりBimodal予測器を，クラスEvaGshareによりGshare予測器を，クラスEvaBimodeによりBimode予測器を評価する．

2.5.3 クラス EvaBimodal

Bimodal 予測器を評価するためのクラス EvaBimodal のコードを示す．このコードによりBimodal予測器のパラメータnを12から20まで変化させた9個の構成のデータを同時に得ることができる．

```
class EvaBimodal{
  int n;
  bimodal **b;
public:
  EvaBimodal();
  ~EvaBimodal();
  void step(int, uint64_t);
};

EvaBimodal::EvaBimodal(){
  n=9;
  b = new bimodal*[n];
  for(int i=0; i<n; i++)
    b[i] = new bimodal(12+i);
}

EvaBimodal::~EvaBimodal(){
  for(int i=0; i<n; i++) delete b[i];
}

void EvaBimodal::step(int r, uint64_t pc){
  int p; /* predicted value */
  for(int i=0; i<n; i++){
    b[i]->predict(pc, &p);
    b[i]->regist(pc, r, p);
  }
}
```

同様に，Gshare 予測器を評価するクラス EvaGshare と，Bimode予測器を評価するクラスEvaBimodeが存在するが，これらのコードは省略する．

3 評価

本章では，機能レベルのシミュレータSimAlpha[7]を用いて，実装した分岐予測器の性能を評価する．SimAlphaは，ライブラリを静的にリンクしたアプリケーションのコードを実行する．オペレーティングシステムのコードが評価に含まれていない点に注意する必要がある．また，YAGS予測器[2]の評価でおこなわれているようなコンテキストスイッチの影響は考慮していない．すなわち，一つのアプリケーションプログラムの評価が全て終了した後に，次のプログラムの評価を開始する．

評価には、SPEC CINT95[1]の8本のプログラムを利用する．シミュレーションの命令数を抑えるように入力パラメタを調整する．SPEC CINT95のバイナリはDEC Cコンパイラ，最適化オプションO4を用いて生成する．

表 1: ベンチマークプログラム SPEC CINT95 の実行命令数と分岐命令の実行頻度

Program	実行命令数	分岐命令
099.go	138 million	12.4%
124.m88ksim	127 million	10.2%
126.gcc	150 million	15.5%
129.compress	142 million	9.3%
130.li	208 million	15.8%
132.ijpeg	172 million	6.1%
134.perl	153 million	14.4%
147.vortex	184 million	12.3%

各ベンチマークの実行命令数を表1の2列目にまとめる．各ベンチマークの分岐命令の実行頻度を3列目にまとめる．Alpha AXP アーキテクチャにおける BR(Unconditional Branch) 命令，BSR(Branch to Subro utine) 命令を含む分岐命令を評価対象とする．ジャンプ命令は評価の対象

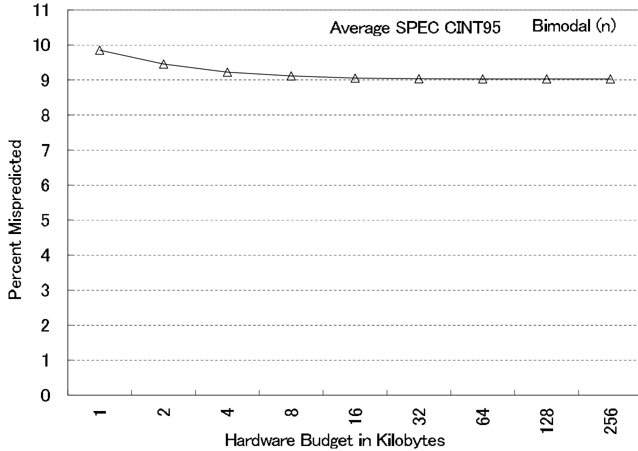

図 6: Bimodal 予測器の評価結果

外である．

3.1 Bimodal 予測器の評価結果

Bimodal 予測器の評価結果を図 6 に示す．縦軸は予測ミス率であり，下に行くほど分岐予測器として性能が高いことになる．横軸はハードウェア量である．図 2 に示したパラメータ n の値を 12 から 20 まで変化させて測定した結果をまとめた．この時のハードウェア量は 1KB から 256KB に対応する．SPEC CINT95 の 8 本のベンチマークプログラムの平均予測ミス率を示した．

図 6 の結果から，ハードウェア量を増やすことで，Bimodal 予測器のミス率が単調に減少することがわかる．ただし，ハードウェア量を 8KB 程度に増やしたところでミス率が飽和する．この時のミス率は約 9%であり，予測精度は悪い．

3.2 Gshare 予測器の評価結果

Gshare 分岐予測には，図 3 に示した様に，PHT のエントリ数を指定するパラメータ n と，分岐履歴レジスタのビット長を指定する m という 2 つのパラメータが存在する．

評価においては，n の値を 12 ビットから 20 ビットまで変化させる．この時のハードウェア量は 1KB から 256KB に対応する．また，取りうる可能性のある分岐履歴レジスタの全てのビット長の予測精度を測定し，その中で最も高い性能を示した構成をまとめる．この構成を，分岐履歴レジスタのビット長が最適という意味で Gshare.best と記述する．Gshare.best のパラメータ n と m の値を表 2 にまとめる．

Gshare.best の評価結果を図 7 に示す．縦軸は予測ミス率であり，下に行くほど分岐予測器として性能が高いことになる．横軸はハードウェア量である．SPEC CINT95 の 8 本のベンチマークプログラムの平均予測ミス率を示した．

表 2: Gshre.best の最適な分岐履歴レジスタのビット長

n	12	13	14	15	16	17	18	19	20
m	4	8	9	10	12	13	15	16	19

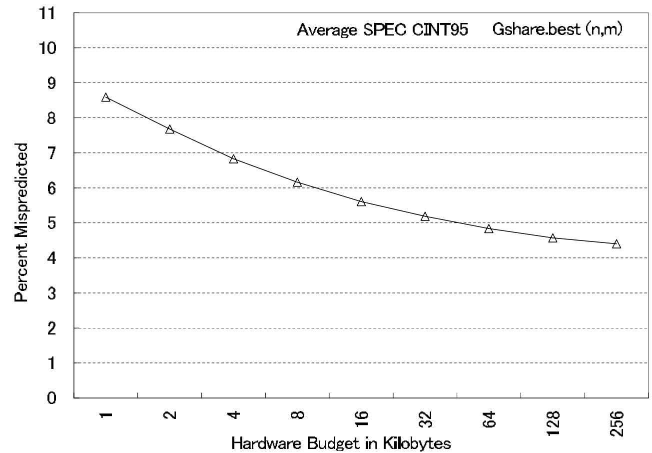

図 7: Gshare.best の評価結果

図 7 の結果から，ハードウェア量を増やすことでミス率が単調に減少することがわかる．Bimodal 予測器とは異なり，測定した範囲ではハードウェア量の増加に伴い，ミス率が減少する．ハードウェア量が 256KB の時の予測ミス率は 4.4%に達する．

3.3 Bimode 予測器の評価結果

Bimode 分岐予測には，図 5 に示した様に，Direction PHT のエントリ数を指定するパラメータ n と，分岐履歴レジスタのビット長を指定する m と，Choice PHT のエントリ数を指定するパラメータ s という 3 つのパラメータが存在する．

評価においては，n の値を 11 ビットから 19 ビットまで変化させる．s の値は n と等しい，すなわち Direction PHT と Choice PHT のエントリ数を等しいとする．この時のハードウェア量は 1.5KB から 384KB に対応する．また，取りうる可能性のある分岐履歴レジスタの全てのビット長の予測精度を測定し，その中で最も高い性能を示した構成をまとめる．この構成を，分岐履歴レジスタのビット長が最適という意味で Bimode.best と記述する．

表 3: Bimode.best の最適な分岐履歴レジスタのビット長

n, s	11	12	13	14	15	16	17	18	19
m	9	12	13	14	15	16	17	18	19

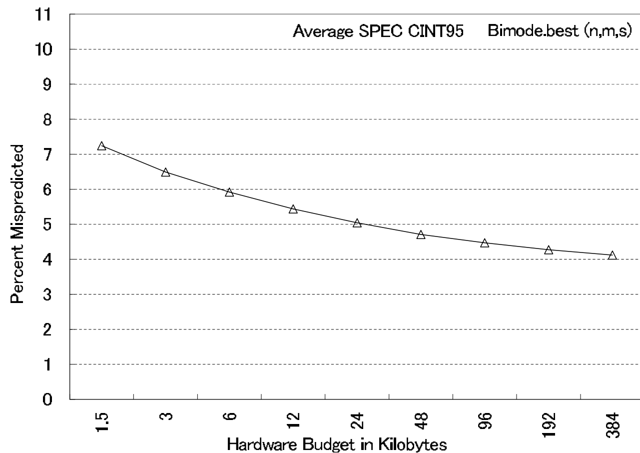

図 8: Bimode.best の評価結果

Bimode.best のパラメータ n と m の値を表 3 にまとめる．

Bimode.best の評価結果を図 8 に示す．縦軸は予測ミス率であり，下に行くほど分岐予測器として性能が高いことになる．横軸はハードウェア量である．SPEC CINT95 の 8 本のベンチマークプログラムの平均予測ミス率を示した．

図 8 の結果から，ハードウェア量を増やすことでミス率が単調に減少することがわかる．Gshare.best と同様に，測定した範囲ではハードウェア量の増加に伴い，ミス率が減少する．ハードウェア量が 384KB の時の予測ミス率は 4.1%に達する．

3.4 分岐予測器の比較

Bimodal 予測器，Gshare.best，Bimode.best の予測精度を比較する．評価結果を図 9 にまとめる．縦軸は予測ミス率であり下に行くほど性能が高い．横軸は対数で示したハードウェア量である．

3 つの分岐予測器を比較した図 9 の結果から，Bimodal 予測器の予測精度が最も悪いこと，全てのハードウェア量の構成において Bimode.best が最も高い予測精度を示すことを確認できる．

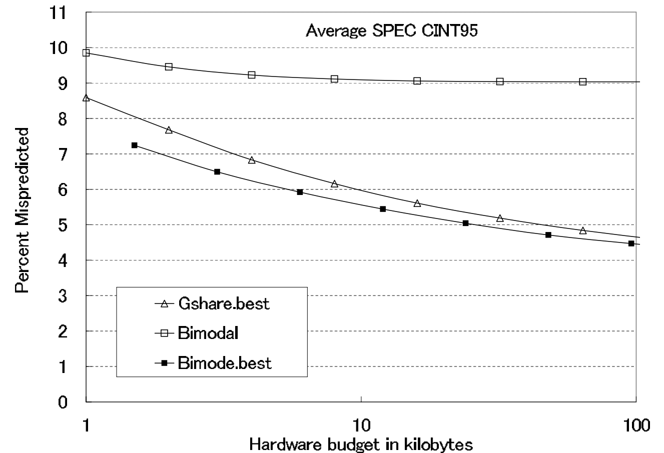

図 9: 分岐予測器の比較

4 おわりに

命令発行幅の増大と命令パイプライン長の増大により，プロセッサ性能に与える分岐予測器の重要性が増している．このような中，過去数十年の間に，予測精度の向上を目指して様々な分岐予測器が提案されてきた．

本プロジェクトの目的は，高級言語を用いた実装を示すことで代表的な分岐予測器の挙動を明確にすること．及び，ベンチマークプログラムを用いて代表的な分岐予測器の性能を評価することにある．

高性能プロセッサのための代表的な分岐予測器として，Bimodal 分岐予測器，Gshare 分岐予測器，Bimode 分岐予測器を取り上げ，これらの分岐予測器の実装を示した．また，ベンチマークプログラム SPEC CINT95 を用いて実装した分岐予測器を評価し，Bimode 予測器が最も高い性能を示すことを確認した．

SimAlpha Version 1.0 と、分岐予測器の性能を評価するために変更を加えたソースコードは次の URL からダウンロードできる

`http://www.yuba.is.uec.ac.jp/~kis/SimAlpha/`

参考文献

[1] Standard Performance Evaluation Corporation. SPEC benchmark suites. http://www.spec.org.

[2] A. N. Eden and T. Mudge. The yags branch prediction scheme. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pp. 69–77, 1998.

[3] Chih-Chieh Lee, I-Cheng K. Chen, and Trevor N. Mudge. The bi-mode branch predictor. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pp. 4–13, 1997.

[4] S. McFarling. Combining Branch Predictors. Technical report, WRL Technical Note TN-36, Digital Equipment Corporation, 1993.

[5] James E. Smith. A study of branch prediction strategies. In *Conference proceedings of the eighth annual symposium on Computer Architecture*, pp. 135–148, 1981.

[6] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pp. 51–61, 1991.

[7] 吉瀬謙二, 本多弘樹, 弓場敏嗣. Simalpha: C++で記述したもうひとつの alpha プロセッサシミュレータ. 情報処理学会研究報告 2002-ARC-149, pp. 163–168, August 2002.

[8] 斉藤史子, 山名早人. 投機的実行に関する最新技術動向. 情報処理学会研究報告 2001-ARC-145, pp. 67–72, November 2001.

[9] 斉藤史子, 北村建志, 山名早人. ハイブリッド分岐方向予測機構の性能比較. 情報処理学会研究報告 2002-ARC-150, pp. 89–94, November 2002.

[10] 野口良太, 森敦司, 小林良太郎, 安藤秀樹, 島田俊夫. 分岐方向の偏りを利用し破壊的競合を低減する分岐予測方式. 情報処理学会論文誌, Vol. 40, No. 5, pp. 2119–2131, May 1999.